PhD-FSTC-2016-21
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 24 June 2016 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

DENNIS APPELT
Born on 25 October 1984 in Saarbrücken (Germany)

AUTOMATED SECURITY TESTING OF WEB-BASED SYSTEMS AGAINST SQL INJECTION ATTACKS

DISSERTATION DEFENSE COMMITTEE

PROF. DR. ING. LIONEL BRIAND, Dissertation Supervisor
University of Luxembourg (Luxembourg)

DR. JACQUES KLEIN, Chairman
University of Luxembourg (Luxembourg)

DR. CU DUY NGUYEN, Deputy Chairman
University of Luxembourg (Luxembourg)

PROF. DR. ALEXANDER PRETSCHNER, Member
Technical University of Munich (Germany)

PROF. DR. MARCO VIEIRA, Member
University of Coimbra (Portugal)

Abstract

Injection vulnerabilities, such as SQL Injection (SQLi), are ranked amongst the most dangerous types of vulnerabilities. Despite having received much attention from academia and practitioners, the prevalence of SQLi is common and the impact of their successful exploitation is severe. In this dissertation, we propose several security testing approaches that evaluate web applications and services for SQLi vulnerabilities and common IT infrastructure components such as Web Application Firewalls for their resilience against SQLi attacks. Each of the presented approaches covers a different aspect of security testing, e.g. the generation of test cases or the definition of test oracles, and in combination they provide a holistic approach.

The work presented in this dissertation was conducted in collaboration with SIX Payment Services (formerly CETREL S.A.). SIX Payment Services is a leading provider of financial services in the area of payment processing, e.g. issuing of credit and debit cards, settlement of card transactions, online payments, and point-of-sale payment terminals. We analyse the challenges SIX is facing in security testing and base our testing approaches on assumptions inferred from our findings. Specifically, the devised testing approaches are **automated**, applicable in **black box** testing scenarios, able to **assess and bypass Web Application Firewalls**, and use an **accurate test oracle**. The devised testing approaches are evaluated with SIX' IT platform, which consists of various web services that process several thousand financial transactions daily.

The main research contributions in this dissertation are:

- An assessment of the impact of Web Application Firewalls and Database Intrusion Detection Systems on the accuracy of SQLi testing.
- An input mutation technique that can generate a diverse set of SQLi test cases. We propose a set of mutation operators that are specifically designed to increase the likelihood of generating successful SQLi attacks.
- A testing technique that assesses the attack detection capabilities of a Web Application Firewall by systematically generating SQLi attacks that try to bypass it.
- An approach that increases the attack detection capabilities of a Web Application Firewall by inferring a filter rule from a set of bypassing SQLi attacks. The inferred filter rule can be added to the Web Application Firewall's rule set to prevent attacks from bypassing.
- An automated SQLi test oracle that is designed to meet the specific requirements of SQLi testing in an industrial context and that is independent of any specific SQLi test case generation technique.

Acknowledgements

Over the last years, a great many people contributed to the success of my PhD project. I would like to express my gratitude to all those who helped me succeed with my work.

I would like to thank my supervisor, Prof. Lionel Briand, for his significant effort to make my research work match the highest academic standards. I am grateful to have had the opportunity to work with and learn from one of the best researchers in the field.

I would like to thank my co-supervisor, Dr. Cu Duy Nguyen, for all his encouragement throughout my PhD project and his advice on how to conduct good research. His support was undoubtedly crucial for the success of my PhD project.

I would like to thank Prof. Alexander Pretschner, for his inspiring lectures and for encouraging me in the first place to pursue doctoral studies.

I would like to thank SIX Payment Services (formerly Cetrel S.A.) for the numerous meetings, in which the engineers detailed SIX' IT systems, and for providing the case study system that was the subject of my empirical studies. In particular, I would like to thank Xavier Stenuit, Joffrey Nutin and Alain Barthelemy for providing me with technical assistance and guidance related to the case study system.

I would like to express my gratitude for all the friendships I have formed with my colleagues in the Software Verification and Validation Lab. My colleagues not only provided me with helpful advice along the way but also contributed to a warm work environment.

Finally, I would like to thank my mother and father for their lifelong support and for enabling me to pursue my dreams.

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR4800382).

Contents

List of Figures

List of Tables

List of Definitions

Glossary

$\mu 4SQL$ A set of input mutation operators for the generation of SQLi test cases [Appelt et al., 2014].

AntiSQL AntiSQL is a tool provided by the vendor of the SQL parser we use in our work, which takes log files containing SQL statements as inputs, and reports whether their content is classified as attacks or legitimate statements.

Burpsuite A commercial security testing tool suite available at http://portswigger.net/burp. It has a vulnerability scanner that targets many types of vulnerabilities, including those in the OWASP top 10 [Williams and Wichers, 2013]. For detecting SQLi, Burpsuite (version 1.6.23) has a fixed list of 134 built-in SQLi test payloads.

Cyclos Cyclos is a popular open-source Java/Servlet web application for e-commerce and online payment (http://project.cyclos.org).

GreenSQL GreenSQL is a popular database security solution for controlling database accesses, blocking SQLi attacks, among other features. It intercepts communications between applications and databases, learns patterns of regular SQL statements, and then, blocks malicious statements from getting to databases under protection.

Hotel Reservation Service The Hotel Reservation Service is a service-oriented appliation providing web services for hotel room reservation. It was developed and used by Coffey et al. [Coffey et al., 2010a].

ML-Driven A machine learning-driven approach to testing web application firewalls [Appelt et al., 2015].

ML-Driven B A variant of ML-Driven that is more likely to explore the input space than to focus on a few promising areas in the search space.

ML-Driven D A variant of ML-Driven that is more likely to focus on a few promising areas in the search space than to explore the input search.

ML-Driven E An enhanced variant of ML-Driven that balances the sampling of the input search between exploration and exploitation..

ModSecurity ModSecurity is an open-source Web Application Firewall.

Payment Card Industry Data Security Standard The Payment Card Industry Data Security Standard is a compliance standard that dictates rules for the secure storage and processing of credit card data. All companies that handle credit card data have to prove their compliance with the standard in regular periods.

RAN A random attack generation approach to testing web application firewalls.

Security Oracle for SQLi Attacks (SOFIA) is a test oracle that is tailored for the specific requirements of SQLi testing. See Chapter 8 for a detailed description.

SqlMap A popular state-of-the-art open source tool for penetration testers to detect and exploit SQLi vulnerabilities (available at http://sqlmap.org). It supports various database management systems and implements many heuristics to generate test payloads for different types of SQLi.

SugarCRM SugarCRM is a Customer Relationship Management System written in PHP (http://www.sugarcrm.com).

Taskfreak A web application written in PHP for project management available at http://www.taskfreak.com.

TheOrganizer A web application that supports management and organisation of the activities in a personal agenda (http://www.apress.com/9781590596951). The application is written in Java (using Servlets, J2EE and Spring JDBC).

Wordpress Wordpress is a popular blogging and news publishing platform written in PHP (https://wordpress.org).

Xavier A tool for the automated testing of web services for SQLi vulnerabilities [Appelt et al., 2014, Appelt et al., 2015]. Powered by a grammar developed specifically for SQLi attacks and machine learning, Xavier can generate diverse test payloads that can bypass web application firewalls and detect SQLi vulnerabilities. See Chapter 5 and Chapter 6 for a detailed description.

Acronyms

API Application Programming Interface.

DIDS Database Intrusion Detection System.

GA Genetic Algorithm.

HPC High Performance Cluster.
HRS Hotel Reservation Service.
HTML Hypertext Markup Language.
HTTP Hyper Text Transfer Protocol.
HTTPS HTTP over TLS.

IDS Intrusion Detection System.

LDAP Lightweight Directory Access Protocol.

OWASP Open Web Application Security Project.

PCI-DSS Payment Card Industry Data Security Standard.
PHP Hypertext Preprocessor.

SOAP Simple Object Access Protocol.
SOFIA Security Oracle for SQLi Attacks.
SQL Structured Query Language.
SQLi SQL Injection.

WAF Web Application Firewall.
WSDL Web Service Description Language.

XML Extensible Markup Language.
XPATH Extensible Markup Language (XML) Path Language.

Chapter 1

Introduction

1.1 Context

In recent years, the World Wide Web evolved from a static source of information to an important application platform. Banking, shopping, education, social networking and even government processes have become available through the web. The rise of cloud-backed applications and web-centric operating systems like Windows 10 or ChromeOS further accelerated this shift.

The popularity of web applications can be attributed to their availability, accessibility and flexibility. However, this also caused the web to become the target of malicious attackers. Recent studies found that the number of reported web vulnerabilities is growing sharply [Fossi and Johnson, 2009]. Web applications experience on average 27 attacks per hour and as many as 24158 attacks per hour at peaks because of attack automation [Beery and Niv, 2013]. Amongst the various types of attacks, the Open Web Application Security Project (OWASP) ranks injection attacks as the most dangerous attacks, while stating that the impact of injection attacks is severe and their prevalence is common.[1] Injection attacks like SQL Injection (SQLi) exploit poorly validated input fields to inject code fragments that can cause the application to behave in an unintended way or expose sensitive data. The concept of SQLi vulnerabilities was first described in 1998 [Forristal, 1998] and has since received much attention from academia as well as practitioners, yet SQLi incidents occur on a frequent basis since developers work under pressure and are not always fully aware of injection issues, and web applications and services are increasingly complex.

In this work, we study the challenges of security testing in an industrial context and try to understand why SQLi is still prevalent despite being well studied. Based on our analysis, this dissertation presents several complementary security testing approaches for web applications and services. The work has been done in collaboration with SIX Payments Services (formerly CETREL S.A.), a leading financial service provider in Luxembourg and Switzerland. The devised testing strategies are motivated by the challenges SIX is facing in security testing and are evaluated with SIX' IT platform, which consists of various web services that process several thousand financial transactions daily.

SIX offers a range of web services to its institutional clients, e.g. international banks and merchants, for the management of payment cards and handling of financial transactions. Since the web

[1] For details refer to: https://www.owasp.org/index.php/Top_10_2013-A1-Injection

services are accessible via the Internet, SIX has a strong interest to secure their IT platform from malicious users. Typically, credit card data and financial transactions are a valuable target for hackers and therefore SIX invests considerable efforts to keep its platform and data secure. We developed in collaboration with SIX several security testing approaches to perform vulnerability assessments of their web services and to increase the overall resilience against cyberattacks.

According to the software engineering body of knowledge, software testing consists of the *dynamic* verification that a program provides *expected* behaviours on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain [Pierre Bourque, 2014]. More specifically, *software security testing* is an activity to measure the quality of security relevant properties of a program or service. Examples of security relevant properties are confidentiality, integrity, availability, authentication, authorization, and nonrepudiation [Felderer et al., 2015].

This dissertation presents several approaches that are specifically designed to tackle the challenges of security testing in an industrial context and are evaluated with our collaborator's web services. Since the web services of our partner rely heavily on relational databases, we target SQLi vulnerabilities, one of the most common and severe type of vulnerabilities according to OWASP. The presented testing approaches are applicable in black box testing scenarios, i.e. in scenarios where the source code of the subject application can not be analysed or modified for the purpose of testing, which is a common scenario in companies like SIX. Furthermore, the presented testing approaches aim to achieve a high degree of automation in order to avoid that tedious manual tasks become a bottleneck and to minimize human error. Each of the presented approaches covers a different aspect of SQLi testing, e.g. test case generation or the definition of test oracles, which provide in combination a holistic approach to SQLi testing.

1.2 Research Contributions

In this dissertation, we investigate the challenges of testing web applications and services for SQLi in an industrial context and we devise several testing approaches that tackle the identified challenges. Specifically, we make the following contributions:

- In collaboration with our industrial partner we identified challenges in security testing that might limit the applicability or effectiveness of state-of-the-art testing techniques in an industrial context. The devised testing techniques in this dissertation mainly address these identified challenges. This is covered in Chapter 3.
- A technique that leverages database intrusion detection systems to increase the accuracy of SQLi test oracles. Furthermore, our proposed technique considers the influence of Web Application Firewalls (WAFs) on the results of penetration testing, which are a common building block in IT environment of financial service providers. This contribution has been published [Appelt et al., 2013] and is discussed in Chapter 4.
- An input mutation technique that can generate a diverse set of test cases to evaluate web services for SQLi vulnerabilities. We propose several mutation operators that are specifically designed to increase the likelihood of generating successful SQLi attacks. This contribution has been published as conference paper [Appelt et al., 2014] and Chapter 5 introduces our approach.
- A testing technique that systematically assesses the attack detection capabilities of WAFs. Our technique uses machine learning to identify attack patterns that are likely to be missed by a

WAF under test and systematically exploits such patterns to generate attacks that are likely to bypass the WAF. This contribution has been partly published as conference paper [Appelt et al., 2015] and is currently under review at the journal *IEEE Transactions on Software Engineering*. Chapter 6 presents the approach.
- A technique that improves the attack detection capabilities of a WAF. Given a set of attacks that are not correctly identified by a WAF, this technique infers a filter rule that matches the missed attacks and can in turn be added to the WAF's rule set.
- A SQLi test oracle that is designed to meet the specific requirements of SQLi testing in an industrial context and that is independent of any specific SQLi test case generation technique. This contribution has been submitted to the *International Conference on Automated Software Engineering* (ASE 2016) and is currently under review.

1.3 Dissertation Outline

Chapter 2 presents the foundations and concepts that are used throughout this dissertation. Section 2.1 introduces SQLi, a common and severe type of attack on web applications and services. Section 2.2 introduces protection mechanisms that aim to prevent the exploitation of SQLi vulnerabilities and are typically used in corporate IT environments.

Chapter 3 presents an overview of the challenges in security testing that we identified in collaboration with our industry partner. The testing techniques presented in this work are based on the identified security testing challenges.

Chapter 4 proposes to consider WAFs and Database Intrusion Detection Systems (DIDSs) as parts of the security testing process. We present a testing technique that uses DIDS to increase the accuracy of state-of-the-art penetration testing tools, in particular if the subject under test is protected by a WAF.

Chapter 5 combines our proposed technique from Chapter 4 with an input generation approach that can generate a diverse set of SQLi attacks. In contrast to the previous chapter, our proposed input generation approach generates SQLi attacks that are likely to result in syntactically correct, malicious Structured Query Language (SQL) statements and, thus, is effective at detecting exploitable SQLi vulnerabilities.

Chapter 6 presents a testing technique to assess the attack detection capabilities of a WAF. The technique uses machine learning algorithms to learn attack patterns, which are likely not to be detected by a WAF, and systematically exploits these patterns to generate SQLi attacks.

Chapter 7 presents a technique to improve the attack detection capabilities of a WAF. Starting from a set of missed attacks, our technique infers a filter rule that can be added to a WAF's rule set in order to identify the previously missed attacks.

Chapter 8 presents a test oracle that is specifically designed to meet the requirements of testing for SQLi vulnerabilities in an industrial context.

Chapter 9 summarises the presented testing techniques and discusses future work.

Appendix A provides a detailed analysis of two variants of the WAF testing technique (ML-Driven

D and ML-Driven B) presented in Chapter 6. We used the insights gained in this analysis to devise an enhanced variant of our WAF testing technique (refer to ML-Driven E).

Appendix B provides detailed evaluation results for the WAF testing technique presented in Chapter 6.

Chapter 2

Foundations

2.1 SQL Injection

In this section, a precise definition for each of the terms related to SQLi testing, which will be used throughout this dissertation, is given. Understanding the meaning of terms such as vulnerable, detectable and exploitable might seem intuitive; however, their precise definition might influence the interpretation of results and evaluation of testing techniques.

In most cases, when input values are used in SQL statements, their values are used as data and not as part of the SQL code to be executed. For example, the following Hypertext Preprocessor (PHP) code forms a SQL statement by concatenating a string literal with the user-provided input variable `$country`:

```
$sql="Select * From hotelList where country ='".$country."'";
$result = mysql_query($sql) or die(mysql_error());
```

The depicted SQL statement returns a list of hotels. The user-provided value of the input parameter `$country` is intended to limit the result list to hotels of a specific country. However, if this input is not properly validated and checked for malicious values, a user can provide an input such as:

```
' ; drop table hotelList;--
```

The result of concatenating this input with the previous SQL statement would be:

```
"Select * From hotelList where country ='' ;
        drop table hotelList;--';
```

When the database server executes this SQL code, the `Select` statement would return no values while the `drop table` statement would delete the table hotelList (if permission to drop tables is not configured correctly at the database level to prevent such actions). The rest of the command will not be

executed because it is commented out (– symbol). The input in this case was interpreted as SQL code rather than data, allowing the user to alter the database causing loss of information and unavailability of the system because the table was deleted. We define an SQLi vulnerability as follows:

2.1 Definition: SQL Injection Vulnerability. *A SQLi vulnerability is caused by a block of code statements where potentially malicious input data can be interpreted as SQL code instead of being treated as data.*

Data is classified as untrusted if it is received from an external source. For example, the input values in a user-provided web service request are typically considered to be untrusted data. Whether all or part of the untrusted data is interpreted as SQL code depends on the attack string used and the logic of the application. For example, in the previous SQL statement if the attack string was `Luxembourg'; drop table hotelList;–` then part of the input value (`Luxembourg`) is treated as data while the rest (`drop table hotelList;–`) is interpreted as SQL code.

In the context of black box testing, the statement that causes a vulnerability can typically not be localised. Therefore, we refer instead to an input parameter being vulnerable:

2.2 Definition: Vulnerable SQLi Input Parameter. *A vulnerable SQLi input parameter is the entry point through which potentially malicious data enters an application under test that is subsequently used in an SQLi vulnerability in at least one execution of the system.*

In this work, we use the term *vulnerability* to refer to a vulnerable SQLi input parameter, except if stated otherwise.

The goal of an SQLi testing approach is to detect vulnerabilities. Whether a vulnerability is detected or not also depends on the oracle used. Therefore, we define a detectable vulnerability as follows:

2.3 Definition: Detectable SQL Injection Vulnerability. *An SQLi vulnerability is a vulnerability that can be detected given a specific oracle.*

In some cases, an SQLi vulnerability exists in an application but an attacker might not be able to exploit it. For example, an application might not properly validate an input value that is assumed to be numeric to ensure that it actually contains a numeric value; thus, it is vulnerable to SQLi. However, a WAF might be configured to block any requests containing non-numeric data for the vulnerable input parameter (a concept known as *virtual patching*). Although the application is vulnerable, an attacker would not be able to use the vulnerability to gain any benefit. We can define an exploitable SQLi vulnerability as follows:

2.4 Definition: Exploitable SQL Injection Vulnerability. *An exploitable SQLi vulnerability is a vulnerability that can be used to cause an information leak, an unauthorised change in the state of the database or system, the database to be unavailable or any other unintended and harmful interaction.*

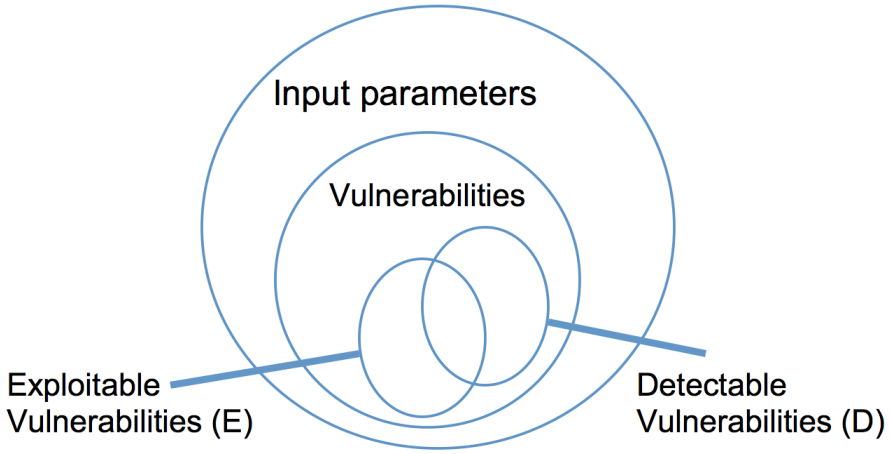


Figure 2.1. Venn diagram illustrating the relationship between the different input parameter classifications.

In some cases, deciding if a vulnerability is exploitable can not be done automatically and requires manual inspection by engineers who have the domain knowledge to decide if a vulnerability is exploitable. For example, a vulnerability in the system might be exploited to leak information but the information that is leaked is not sensitive or can be obtained by any user through alternative methods. However, an automated approach might be able to estimate the probability of a vulnerability being exploitable based on some heuristics. Vulnerabilities can then be ranked based on this probability to reduce the time required for manual inspection or focus efforts on vulnerabilities that might pose a higher threat.

Figure 2.1 illustrates the relationship between the different classifications of input parameters. A subset of all input parameters might be vulnerable, while a subset of those vulnerabilities is exploitable. Detectable vulnerabilities could either be exploitable or not exploitable. The intersection of detectable vulnerabilities and exploitable vulnerabilities ($E \cap D$) is the set of critical security faults that the testing process can find. The set of exploitable vulnerabilities that are not detectable ($E - D$) represents the false negatives of the testing process.

2.2 Runtime Protection Mechanisms of Web Applications

Since an application can in general not be guaranteed to be free of vulnerabilities, several protection mechanisms are available that guard applications at runtime by shielding them from attacks. Having such protection mechanisms in place reduces the risk of a potentially vulnerable application from being exploited. This is in particular important in companies that depend on the secure and reliable functioning of their IT systems and, thus, such companies frequently employ application protection mechanisms (consider the example of our industry partner described in Chapter 3).

Several types of runtime protection mechanisms are used to protect against SQLi attacks. Two commonly used security mechanisms in practice are Web Application Firewalls (WAFs) and Database Intrusion Detection Systems (DIDSs). Since both of these mechanisms can have a significant impact on security testing and play a major role throughout this work, Section 2.2.1 introduces WAFs and Section 2.2.2 introduces DIDSs.

2.2.1 Web Application Firewalls

Firewalls are commonly used to protect hosts in a private network from malicious request originating from outside the network. Over the years, several different types of firewalls were developed. The first generation of firewalls operate on the network layer by examining network packets, e.g. for source and destination ports and addresses. Typically, the inspection of packets is done according to some rule set, i.e. a network firewall forwards a packet to its destination if it complies with the rule set or drops a packet otherwise. For example, for a network firewall that protects a web server it might be desirable to drop any incoming packets whose destination port is not 80 (HTTP) or 443 (HTTPS). By doing so, the firewall effectively protects internal network services on other ports than 80 and 443, which were in particular in the early days of the Internet often vulnerable, from external attackers.

With the rising popularity of web applications and services the attacker's focus shifted from vulnerabilities in network services to vulnerabilities in web-based systems. To exploit such vulnerabilities, an attacker sends specially crafted HTTP requests, which target input sanitization bugs or other flaws in the web application code (for example SQLi). Network firewalls are not effective in stopping attackers from exploiting application-level vulnerabilities, since legitimate and malicious requests use the same destination port and, thus, are typically not distinguishable for network firewalls. To defend from this new threat, a new generation of application-protocol aware firewalls was introduced. For example, for web applications and services, HTTP Firewalls were introduced, which are nowadays commonly known as Web Application Firewalls.

A WAF examines every HTTP/S request submitted by the user to the application to decide if the request should be accepted (if the request is legitimate) or rejected (if the request is malicious). The WAF makes this decision by examining each input value in the request and checking if the value matches the expected format (whitelist filtering) or if the value matches a known attack pattern (blacklist filtering). Typically, this matching is performed according to a set of rules (i.e. regular expressions). For example, consider the following blacklist rule from a popular open-source rule set:[1]

```
/\*!?|\*/|['';]--|--[\s\r\n\v\f]
      |(?:--[^-]*?-)
      |([^\-&])#.*?[\s\r\n\v\f]|;?\\x00
```

The depicted regular expression matches SQL comments, e.g. /**/ or #, which are frequently used in SQLi attacks and, thus, requests containing this strings might be blocked. Similarly, a whitelist rule for a parameter that is expected to contain a credit card number checks if a received input conforms to the format of a credit card number, e.g. a string consisting out of 16 to 19 digits, and blocks the request otherwise.

The performance of the WAF and the protection it provides depends on this set of rules. Since these rules are created and maintained manually by the application owner, they might be error-prone and security holes might be introduced by mistake. On the other hand, attackers are continually searching for ways to evade firewalls by using mechanisms such as obfuscation, where semantically

[1] See OWASP Core Rule Set: https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_-Project

equivalent encodings of attack patterns are used that might not be recognized by the WAF. WAFs are commonly used in industry and, for example, using a WAF is necessary to be compliant with the Payment Card Industry Data Security Standard [PCI Security Standards Council, 2013] for systems that process credit cards.

There are several reasons why a WAF may provide insufficient protection, including implementation bugs or misconfiguration. One way to ensure the resilience of a WAF against attacks is to rely on an automated testing procedure that thoroughly and efficiently assesses its attack detection capabilities. This work addresses this challenge for SQL injections, one of the main types of vulnerabilities in practice.

2.2.2 Database Intrusion Detection System

Database Intrusion Detection System (DIDS) are the subject of a large body of literature [Chung et al., 2000, Bertino et al., 2005, Srivastava et al., 2006, Hashemi et al., 2008, Valeur et al., 2005, Santos et al., 2014] and several commercial DIDSs are available (e.g., GreenSQL [GreenSQL LTD., 2013], Oracle Audit Vault and Database Firewall [ora, 2014]). A DIDS monitors the access to a database and is often implemented as a proxy that intercepts the SQL statements sent from the application to the database. DIDSs have an advantage over WAFs in that they have access to the SQL statement after it is formulated and, therefore, have more information to decide if an SQL command is an attack. Vendors claim that this increased visibility results in the accurate detection of illicit database accesses like SQLi. A study compared several intrusion detection tools, which operate on different system levels such as Network, Application, or Database, and finds GreenSQL, a Database Intrusion Detection System, to be the most accurate amongst the compared tools, although far from being perfect [Elia et al., 2010].

Besides intrusion detection, some commercial solutions provide also intrusion prevention mechanisms (e.g. GreenSQL). Such solutions can usually be configured to either a prevention mode, where the execution of malicious statements is blocked, or to a monitoring mode, where suspicious statements are allowed to execute, but logged for further examination by an administrator. For software testing purposes, in particular the latter mode is promising, since preventing a statement from execution might adversely affect the functionality of the application under test, which might potentially lead to false positives.

Typically, DIDSs use either a risk-based or learning-based approach to decide if an SQL statement is malicious. The risk-based approach assigns a risk score to each intercepted SQL statement, which reflects the probability of the statement being malicious. To calculate the risk score, each statement is assessed according to various criteria, e.g. for SQL fragments frequently used in SQLi attacks like the comment sign or a tautology. In the learning-based approach, the security engineer first sets the DIDS to a learning mode and issues a number of legal requests that exercise the application's normal behaviour. The DIDS, thereby, learns the different forms of SQL statements that the application can execute. When the learning is completed, the security engineer sets the DIDS to the detection mode, where any statement that does not comply with the learned statements is flagged as malicious. The effectiveness of the DIDS is dependent on this learning phase: If the requests issued in the learning phase do not represent all legal behaviour of the application, legal requests might be flagged as suspicious when the DIDS is used in practice and this might lead to a high false positive rate.

Chapter 3

Overview of Research Problems and Proposed Solutions

The testing strategies presented in this work were elaborated in cooperation with our industrial partner, SIX Payment Services (formerly CETREL S.A.). SIX Payment Services is a provider of financial services in the area of payment processing, e.g. issuing of credit cards and debit cards, settlement of card transactions, online payments, and point-of-sale payment terminals. We identified several key challenges that SIX is facing with respect to security testing and we believe that these challenges are representative for similar companies. The working assumptions of the testing approaches proposed in this work are in essence derived from these key challenges.

Lack of source code. External companies develop various components of the business critical software at SIX and our partner does not have access to the source code. In consequence, our partner is not able to analyse, instrument, or modify the source code for purposes of software testing. However, although our partner is not the developer of the software, they are responsible for the security of the provided financial services and must ensure that no vulnerabilities within the various components can be used to steal customer funds or to misuse the provided services in any other unintended way. Failure to do so might lead to severe financial losses as can be seen from incidents[1] that occurred at similar companies. For this reason, our partner has to test the software for security vulnerabilities. What complicates security testing in this case is that the source code is not available and thus no details about its inner workings are known.

Black box security testing techniques are usually applied in scenarios where the source code is not available. These techniques typically start by identifying input fields of the application under test, e.g. by processing WSDL files or web crawling. Then, malicious request containing various attack types, e.g. SQLi, are submitted to the application under test and the response is analysed to determine if the attack was successful. However, recent studies assessed black box testing techniques and found that their effectiveness and accuracy is limited. Khoury et al. [Khoury et al., 2011] conducted a study of several state-of-the-art security scanners and found that this ineffectiveness might be caused by their limited ability to deeply explore an application and their limited ability in analysing responses. A

[1] Interested readers are referred to:
http://www.nytimes.com/2013/05/10/nyregion/eight-charged-in-45-million-global-cyber-bank-thefts.html
http://krebsonsecurity.com/2014/03/thieves-jam-up-smuckers-card-processor/
http://krebsonsecurity.com/2012/03/mastercard-visa-warn-of-processor-breach/

similar study found that while state-of-the-art scanners are efficient in finding low hanging fruit, e.g. reflective cross-side scripting, they fail finding hard to detect vulnerabilities, e.g. local and remote file inclusions [Chen, 2015].

In this work, we address the described issues by developing security testing approaches that do not require access to source code. To increase the effectiveness and accuracy of state-of-the-art black box testing, we propose to observe the interactions of the subject application under test with its environment, e.g. database accesses or network traffic. While it is common for companies like SIX to have no access to the source code, they often own and operate the hardware and network on which the software is executed. By observing, for example, the database connection of the tested application during a test run, we expect to devise more precise test oracles. Chapter 4 introduces the idea of using a commercial database Intrusion Detection System (IDS) as a test oracle. Chapter 5 further develops this idea and combines it with an effective approach for test case generation. Based on our experience of using a commercial database IDS as a test oracle, Chapter 8 introduces a new SQLi test oracle that is tailored for the specific requirements of security testing.

Effort intensive testing. Due to the need to comply with the Payment Card Industry Data Security Standard (PCI-DSS), SIX has to perform security tests on all applications that process credit card data [18]. These tests have to occur at least once a quarter and have to be applied to all running software, no matter if the code or the configuration was changed. Additionally, security tests have to be performed prior to deploying modified applications to production systems. All these testing activities lead to a high workload for the responsible security team. From a business perspective, it is desirable to lower the time and manual labour involved in testing to reduce the costs.

To reduce the manual testing efforts required for PCI-DSS compliance, it is desirable to have techniques that enable a higher degree of automation. Therefore, all approaches developed in this work aim to achieve test automation. In particular, Chapter 5 introduces an automated testing technique to detect SQLi vulnerabilities in web services, Chapter 6 introduces an automated approach to test WAFs and Chapter 7 devises an automated approach to improve attack detection rules.

Lack of effective testing strategies for WAFs. Our industry partner employs several defence mechanisms to increase the resilience of their IT systems against cyberattacks. For example, they set up a WAF to defend against attacks on their web services or a database intrusion detection system to detect illicit data access. While such systems can considerably reduce the risk of successful attacks, their configuration and maintenance can be time consuming and error-prone. As a result, misconfiguration and the lack of time and tools for thoroughly testing configuration changes may result in weakening the defence mechanisms.

We analysed our partner's procedure of configuring and maintaining the WAF and found two key scenarios that reduce the WAF's attack detection capabilities: Firstly, in case the WAF's configuration changes, the validation efforts focus on avoiding false positives, i.e. legitimate requests that are being flagged as attacks, while the attack detection capabilities of the WAF are neglected. This imbalance between testing efforts is mainly due to the lack of suitable attack generation tools that can test WAFs. On the other hand, the IT Security Engineer has an extensive test suite of legitimate requests, which makes testing for false positives straightforward and comparably easy. Secondly, there is a plethora of ways to obfuscate attack strings, some of which the IT Security Engineer is not aware of, and this might lead to incomplete WAF filter rules that do not catch obfuscated attacks.

We address the identified issues by deriving a systematic and automated testing technique for WAFs in Chapter 6. The approach generates a diverse set of attacks and checks if a WAF correctly identifies the attacks. Furthermore, our approach considers a large number of attack obfuscation methods. Chapter 7 introduces an approach to transform attacks, which are not correctly identified by a WAF, into a regular expression. This regular expression can in turn be added to the WAF's configuration to block attacks similar to the missed ones.

Chapter 4

Assessing the Impact of Web Application Firewalls and Database Intrusion Detection Systems on SQLi Testing

While security testing can be effective at finding vulnerabilities, it does not provide guarantees that an application is free of vulnerabilities. To reduce the remaining risk of a successful attack, practitioners typically use, in addition to security testing, several run-time protection mechanisms to protect their applications against attacks, such as WAFs and database IDSs. WAFs monitor the Hyper Text Transfer Protocol (HTTP) traffic received by the application under protection for attack strings while database IDSs monitor the communication between the application and the database for suspicious SQL statements (refer to Chapter 2.2 for details). We believe that these two technologies can be utilised in the security testing process and can also affect the results of evaluating different techniques.

In this chapter, we assess the impact of using WAFs and database IDSs on testing for SQLi vulnerabilities, which are one of the most widely spread types of vulnerabilities [Christey and Martin, 2007, The Open Web Application Security Project (OWASP), 2013]. We propose that WAFs can be used to prioritise vulnerabilities by focusing developers effort on vulnerabilities that are not protected by the WAF. We also investigate the effectiveness and efficiency of using a database IDS as oracle for SQLi testing instead of just relying on the output of the application. We expect that using a database IDS would enhance the detection rates of vulnerabilities.

The results of our case study on two service-oriented web applications with a total of 33 operations and 108 input parameters indicate that using a database IDS as an oracle does indeed improve detection rates. The results also confirm our expectation that it is more challenging to find vulnerabilities when the subject under test is protected by a WAF. In our case study, the used WAF protects many of the vulnerabilities, but however, not all vulnerabilities. Therefore, testing through the WAF can be used to prioritise vulnerability localization and repairing efforts.

The rest of this chapter is organised as follows: Section 4.1 discusses the purpose and implications of using WAFs and database IDSs for security testing. Section 4.2 proposes an SQLi testing approach that uses an exiting DIDS as test oracle. Finally, Section 4.3 presents a case study to evaluate our proposed approach together with a discussion of results and threats to validity.

4.1 Influence of Web Application Firewall and Database Intrusion Detection System on Security Testing

WAFs and DIDSs are common building blocks in corporate IT environments. In fact, in some industry sectors, e.g. payment card processing, a WAF is mandatory for the purpose of compliance [PCI Security Standards Council, 2013]. Given that many companies have such systems already in place, it is worth examining if these systems can be utilized for the benefit of security testing. This section discusses the potential benefits and implications of including WAFs and DIDS in the security testing process, for example an increased vulnerability detection rate, fewer false positives, and vulnerability prioritisation.

4.1.1 Web Application Firewalls

Naturally, using a WAF affects the security assessment of an application; some input parameters might be vulnerable if the application is accessed directly, but not vulnerable when a WAF protects the application. For example, an input parameter that flows to an SQL statement might not be validated for SQLi attack patterns, but the WAF is configured to detect and reject such attack patterns. In some cases in practice, e.g. for legacy systems with known vulnerabilities, all validation and filtering might be delegated to the WAF. Testing applications with such a set-up using approaches that only take into account the application itself and not the WAF might result in determining that all inputs that are used in SQL statements are vulnerable. In consequence, such approaches suffer from a high false positive rate.

Ideally, all vulnerabilities should be patched even if a WAF protects the vulnerable parameter. However, with limited time and resources dedicated to testing, which is often the case in industry, test engineers might want to focus on fixing vulnerabilities that can still be exploited even when using a WAF. Therefore, testing an application for SQLi vulnerabilities through a WAF can be used to identify and prioritise vulnerabilities that can be exploited through the WAF.

Testing through a WAF can also have other useful applications, such as testing the WAF itself. This can be useful, for example, if a choice needs to be made by the application owner between different alternative WAFs. The application can be tested using each WAF and the firewall that provides the most protection can be chosen. Finally, testing the WAF could help in evaluating and refining its rule set. When a vulnerability is found that can be detected while using a WAF, the developers, after fixing the application code to eliminate the vulnerability, can define new rules or adjust existing rules to protect the application against similar types of vulnerabilities. This might be useful to protect the application against similar types of vulnerabilities that might be introduced in subsequent versions of the system.

4.1.2 Database Intrusion Detection System

Existing black box SQLi testing approaches commonly use an oracle that analyses the HTTP output of the application under test to decide if a vulnerability was detected [Antunes and Vieira, 2009, Ciampa et al., 2010, Huang et al., 2003]. In this work, we propose using a database IDS, which is implement as a proxy and intercepts all SQL statements, as an oracle (refer to Chapter 2.2.2 for a description of database IDS). Since such a IDS has access to the SQL statement after all input values have been

inserted in the statement and all processing is done, we expect that using an IDS as an oracle would enhance vulnerability detection rates.

4.2 Implementation

We developed a prototype tool in Java that uses a set of standard attacks as test cases and a state-of-the-art database proxy as an oracle. In particular, the tool is expected to help verify that an oracle that observes database communications to detect SQLi vulnerabilities could improve the detection rate of an SQL testing approach. The architecture of the tool is depicted in figure 4.1. The testing process can be divided into three sub-processes: test case generation, delivery mechanism and vulnerability detection.

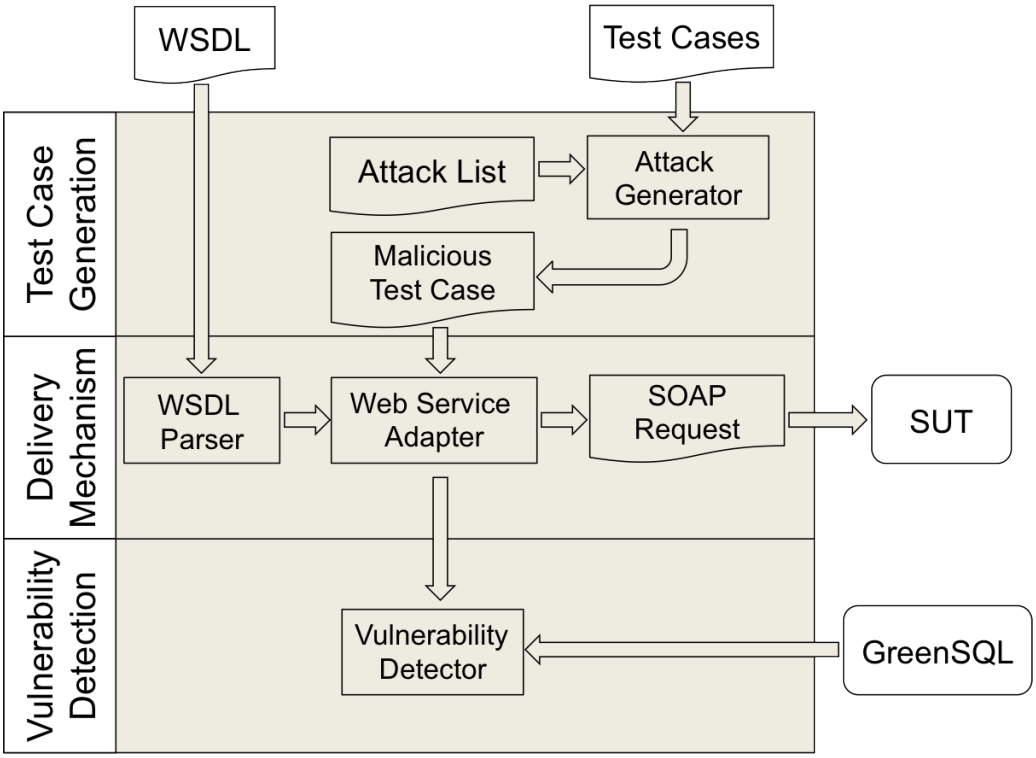


Figure 4.1. Architecture of the prototype testing tool.

The *test case generation* process takes a valid test case as input, i.e. a test case where the input data conforms to the specification of the operation under test. This valid test case is transformed into a malicious test case by replacing one input parameter value at a time with an SQLi attack chosen from a list of standard attacks. We provided the tool with a list of 137 standard attacks that was compiled by Antunes and Vieira [Antunes and Vieira, 2009] and represents common SQLi attacks. A parameter is replaced with an SQL attack from the list until a vulnerability is detected or all attacks are used. This process is repeated for each input parameter of the operation under test. The *delivery mechanism* encapsulates all implementation details that are necessary to deliver the malicious test case to the SOAP-based Web Service and obtain a response. The *vulnerability detection* process uses the state-of-the-art database proxy *GreenSQL* as an oracle to detect vulnerabilities.

GreenSQL is an SQLi attack detection and prevention tool that supports both the learning-based approach and the risk-based approach discussed in Section 2.2.2. In our case study, we used the learning-based approach to detect malicious SQL statements. In brief, in the learning-based mode GreenSQL learns during a training phase all legitimate SQL statements for a database under protection and in the detection phase it blocks all statements that were not learned during training. We chose GreenSQL based on the results of a previous study that compared GreenSQL to five similar tools and found it to be the most effective in detecting SQLi attacks [Elia et al., 2010].

4.3 Evaluation

We designed a case study to answer the following research questions:

RQ1: *What is the impact of using an oracle that observes communications to the database on SQLi vulnerability detection?*

We expect that an oracle that observes the database to determine that a vulnerability was detected might improve the detection rates of an SQLi testing approach compared to an oracle that only relies on the application's output, e.g. HTTP responses. However, using such a database oracle might result in a high number of false positives or have other implications on the results. To answer this question, we conduct an experiment where we compare our prototype tool with SqlMap, a state-of-the-art SQLi testing tool with an oracle based on the HTTP output. We compare the number of vulnerabilities detected and the number of test cases that needed to be generated before the vulnerability was detected. We also examine the requests that detected vulnerabilities for both approaches to investigate whether they led to the formulation of executable malicious SQL statements and, therefore, led to detecting exploitable vulnerabilities.

RQ2: *How does testing the web services directly and testing them through a WAF impact the effectiveness of SQLi testing?*

Generating test cases that are able to detect vulnerabilities in web services through a WAF is naturally expected to be more challenging than testing the application directly, since the WAF provides an additional layer of protection. However, vulnerabilities that can be detected while testing through the WAF pose a more pressing threat since they are completely unprotected. To answer this question, we test the application using the two testing approaches (the state-of-the-art tool and our prototype tool) through a state-of-the-art WAF and compare the results to those obtained without using the WAF. A reduction in the number of vulnerabilities found might indicate that testing through the WAF can be used to prioritise fixing vulnerabilities that are not protected by the WAF. Such reduction might also indicate that we need more advanced test generation techniques for security testing that can penetrate the more sophisticated protection techniques of WAFs and identify harder to detect vulnerabilities.

4.3.1 Subject Applications

We selected web service applications rather than traditional web applications to eliminate the effects of crawling the web application on results. Web services have well-defined and documented APIs that can be used to call the different operations in the application. On the other hand, web applications require a crawling mechanism to be built into the testing technique to explore the application and find input fields that might be vulnerable. The crawling mechanism might impact the effectiveness of the overall testing approach as noted by previous studies [Bau et al., 2010, Doupé et al., 2010, Khoury et al., 2011].

Table 4.1. Details about the two applications we used in the case study

Application	#Operations	#Parameters	LoC
Hotel Reservation Service	7	21	1,566
SugarCRM	26	87	352,026
Total	33	108	353,592

We chose two open-source web service applications as subjects for the case study. Table 5.2 provides information about the number of operations, input parameters and lines of code for the chosen applications. The Hotel Reservation Service (HRS) was created by researchers[1] to study service-oriented architectures and was used in previous studies [Coffey et al., 2010b]. SugarCRM, is a popular customer relationship management system (189.000 downloads in 2013[2]). Both applications are implemented using PHP, use a MySQL database and provide a Simple Object Access Protocol (SOAP) Application Programming Interface (API).

4.3.2 Case Study Set-up

To perform the case study, we conducted two sets of experiments. In the first set of experiments (Figure 4.2), we applied our prototype tool and a state-of-the-art black box security testing tool that relies only on the output of the application to the two case study subjects. We selected *SqlMap* [Damele et al., 2013] as a representative for traditional black box testing tools that rely only on the output. We chose SqlMap because it is an open source free tool that provides support for testing web services as well as web applications. SqlMap is also one of four tools listed on the Open Web Application Security Project (OWASP) website [The Open Web Application Security Project (OWASP), 2013] for automated SQLi testing. The tool was also used in previous studies [Ciampa et al., 2010, Halfond et al., 2009].

In the second set of experiments (Figure 4.3), we applied the same two tools to our case study subjects but tested the applications through a WAF to answer RQ2. We selected *ModSecurity* for our experiments. ModSecurity is a WAF that protects web servers (e.g. Apache, IIS, Nginx) from common threats including SQLi. Since ModSecurity requires a rule set to identify and reject attacks as discussed in Section4.1.1, in our case study we used the OWASP [The Open Web Application Security Project (OWASP), 2013] core rule set (version 2.2.7).

[1] http://uwf.edu/nwilde/soaResources/

[2] http://sourceforge.net

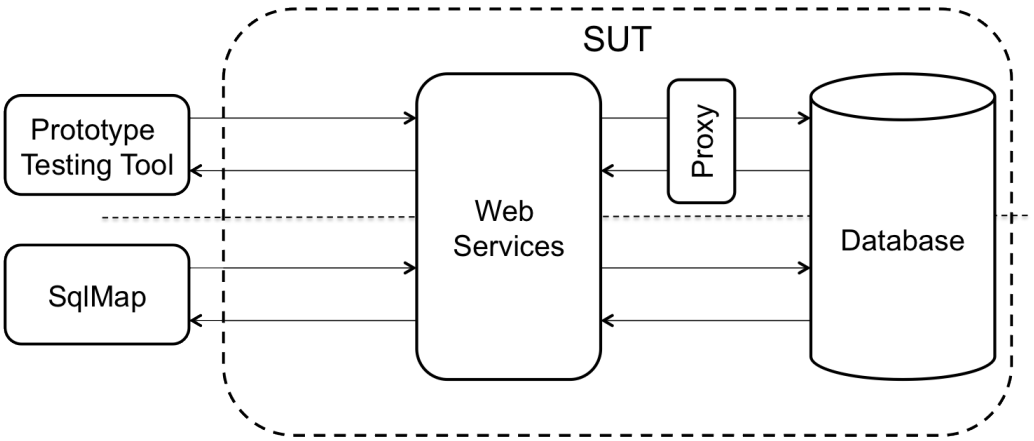


Figure 4.2. Experimental set-up for RQ1: The effect of observing database communications on detection rates.

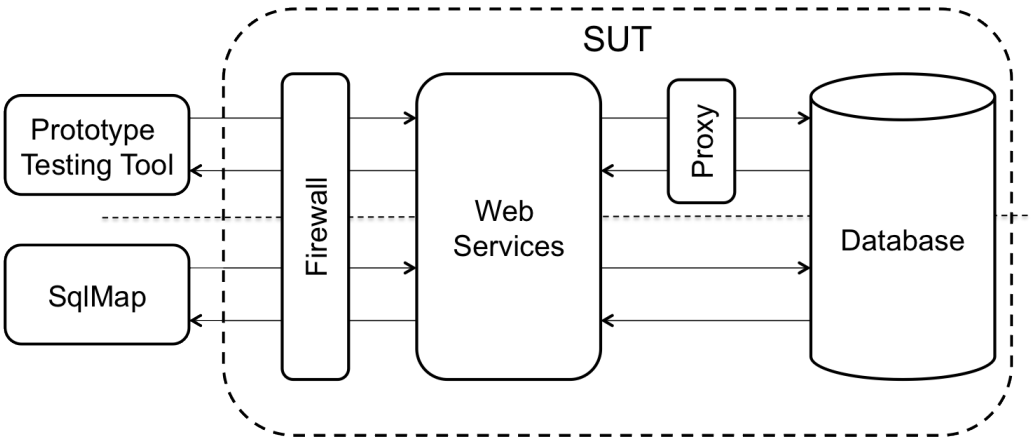


Figure 4.3. Experimental set-up for RQ2: The influence of testing through a Web Application Firewall.

Our prototype tool is deterministic; therefore we ran the tool once on each application and for each set-up. SqlMap, on the other hand, is not deterministic; therefore we ran the tool 30 times for each application and each set-up. We used the default configuration for SqlMap, except for a minor modification that omits test cases that cause the database to pause the execution of a query, for example by calling the `sleep()` operation, to avoid long execution times.

4.3.3 Results

This section discusses the results of running the experiments that we described in the previous section on each of the two case study applications. For each experiment, we counted the number of vulnerabilities found by each tool and the number of tries (requests) that the tool needed to generate and execute before finding each vulnerability. As we mentioned before, SqlMap is non-deterministic, therefore, we repeated any experiments that involve it 30 times and calculated the average number of vulnerabilities and tries.

Table 4.2 summarises all the results obtained from our experiments. Both, the prototype tool and SqlMap, were run on the two web service applications once with a WAF and once without. The results show that the prototype tool, which uses a proxy as an oracle, detects more vulnerabilities

than SqlMap, which relies only on the output of the application, for both applications and using both set-ups (with and without WAF). The only exception is HRS without a firewall where both tools find the same number (6) of vulnerabilities. Manual inspection showed that none of the vulnerabilities reported by either tool is a false positive. We also observe that the number of tries or test cases that the tool needs to execute before detecting the vulnerability (if a vulnerability is found) is significantly higher for SqlMap compared to the prototype tool (1,306.55 vs 6.3 and 566.77 vs 2). These results indicate that using a database proxy as an oracle for SQLi testing could improve detection rates and could also enhance the efficiency of the testing process by detecting vulnerabilities faster.

Table 4.2. Collected data for the described experiments.

Application	Firewall	Prototype		SqlMap	
		#Vulner.	avg. # tries	#Vulner.	avg. # tries
Hotel Reservation Service	without	6	6.3	6	1,306.55
	with	6	28	0	-
SugarCRM	without	6	2	3.87	566.77
	with	3	34	0	-

The difference in the number of detected vulnerabilities when testing with and without a WAF can help us answer RQ2. As we expected, testing through a WAF is more challenging and both testing tools find fewer vulnerabilities in both applications. The only exception is the result of the prototype tool for the Hotel Reservation Service, where the tool found the same number of vulnerabilities with and without a firewall. SqlMap was unable to detect any vulnerabilities for both applications when using a firewall. We also noticed that when vulnerabilities are found, the number of tries needed to find the vulnerabilities also significantly increased (34 vs 2 and 28 vs 6.3). The prototype tool found three vulnerabilities when testing through the WAF for the SugarCRM application. Therefore, these three vulnerabilities are unprotected by the WAF and any debugging or fault repairing effort should be first focused on these three vulnerabilities since the risk of them being exploited is higher. Another conclusion we might draw from these results is that we need more sophisticated test generation techniques and oracles for SQLi testing. SqlMap, a state-of-the-art-tool, was unable to find any vulnerabilities in both applications when using a WAF, while our prototype tool detected six in one application and three out of six in the other. A more sophisticated test generation technique might be able to detect vulnerabilities not found by either tool. Moreover, as hackers are continuously searching for new ways and attack patterns to penetrate WAFs and find security holes in applications, SQLi testing should attempt to emulate these attackers and identify vulnerabilities before the attackers do.

We investigated why the prototype tool finds only three out of six vulnerabilities in SugarCRM in the set-up with a WAF and if there is a difference between the detected and undetected vulnerabilities. Surprisingly, we found that the WAF blocks the valid test cases that are used to test the three missed vulnerable parameters. Recall that our prototype tool uses these valid test cases as a starting point to generate malicious test cases (see Section 5.2) and we assume that the WAF lets these valid test cases pass. However, in our case study this is not for all tested parameters the case and, thus, malicious test cases that are generated from blocked valid test cases are also blocked.

The reason that the valid test cases are blocked is that some of their parameters are formatted as a series of numbers and letters separated by dashes. The rule set of ModSecurity, which is the WAF used in our case study, includes a rule that blocks any request that contains more than five

special characters (e.g., hash signs, quotes, dashes). Some of the valid test cases violate this rule and therefore they are blocked. We expect that this does not happen in practice: Typically, security engineers customise the configuration and rule set of the WAF to ensure that the normal operations and functionality of the application are not affected. This highlights the need for using real industrial case studies when evaluating tools and techniques to obtain more realistic results that reflect what happens in real systems and contexts.

We also examined the test cases that successfully detected vulnerabilities when using our prototype tool. We found that these test cases changed the structure of the SQL statements they affected causing GreenSQL to flag them as SQLi attacks. However, we also found that the resulting SQL statements were not executable because they were syntactically incorrect. For example, one of the attack strings used that detected a vulnerability was `' UNION SELECT`. The vulnerable SQL statement was:

```
$sql="Select * From hotelList where country ='".$country."'";
```

The SQL statement after injecting the variable `$country` with the attack string `' UNION SELECT` would be:

```
Select * From hotelList where country =' ' UNION SELECT'
```

GreenSQL will detect this statement as an SQLi attack since the structure of this statement differs from the previously learned statements. However, the statement itself is not executable and would cause the database server to raise a syntax error when attempting to execute the statement. If the resulting SQL statement was syntactically correct and executable, we might be able to have more confidence in that the detected vulnerability is exploitable. If one of the test cases that detected the vulnerability was used by an attacker, he or she would not be able to gain any benefit from the attack. This suggests that we need to enhance the oracle to get more useful results that can help identify not just detectable vulnerabilities but also exploitable vulnerabilities and produce test cases that result in executable SQL statements that change the behaviour of the application. This can be done, for example, by improving the oracle by combining the database proxy with an additional oracle that checks the syntactical correctness of the resulting SQL statement.

4.3.4 Threats to Validity

This section discusses the threats to validity of our results in this study using the standard classification of threats [Wohlin et al., 2000]:

Internal Threats: The internal threats to validity in this study are related to generation of test cases and the stopping criterion of each approach when studying the effect of the test oracle. Both approaches start from a valid test case when testing each web service operation. We used the same initial test cases for both the prototype tool and SqlMap to avoid experimenter bias.

External Threats: The external threats are related to the choice of case study subjects, the SQLi testing approaches and the ability to generalise results. Although we only used two systems in the

case study, one of the two systems is used by real users as the number of downloads indicates. More experiments with different types of systems might be needed before being able to generalise results. Although we only used two approaches to generate test cases, these two approaches are representative of the state of the art in black box testing, as the review of related work indicates.

Construct Threats: We used the number of detected vulnerabilities to measure effectiveness and used the number of test cases generated before a vulnerability is detected to measure efficiency. Detecting vulnerabilities is the goal of any SQLi testing approach, therefore, the number of vulnerabilities seems like the most natural choice to measure effectiveness. The number of requests (or test cases) issued before detecting a vulnerability is a more reliable method of measuring efficiency since execution time might be effected by the environment and/or other processes performed by the CPU while running the experiments.

4.4 Related Work

This chapter briefly reviews existing techniques for black box SQL injection testing and also review the results of empirical studies that compare different black box testing techniques.

Huang et al. [Huang et al., 2003] proposed a black box SQL injection approach that learns the application's behaviour and then compares this to the behaviour of the application when SQL injection attacks are submitted. Antunes and Vieira [Antunes and Vieira, 2009] use a similar oracle but focus on SQL and server errors rather than the whole output. For example, if the legal test case led to an SQL or server error but the attack was successful, the approach infers that a vulnerability was found since the attack was able to circumvent the checks that caused the original error. Ciampa et al. [Ciampa et al., 2010] analyse the output, including error messages, of both legal and malicious test cases to learn more about the type and structure of the back-end database. This information is then used to craft attack inputs that are more likely to be successful at revealing vulnerabilities. These approaches use an oracle that relies on observing and analysing the output, while we propose using a database proxy as an oracle to enhance detection rates.

Fonseca et al. [Fonseca et al., 2014] proposed to evaluate web application security mechanisms, e.g. DIDSs and web application vulnerability scanner, by injecting vulnerabilities into the source code of web applications and automatically attacking them. The DIDS was evaluated by attacking the injected vulnerabilities and by checking if the DIDS identified the attacks. The web application vulnerability scanners were evaluated by testing the web applications and by checking if they find the injected vulnerabilities. In contrast to our work, the subject under test are web applications while Fonseca et al. assess DIDS and web application scanners.

Several empirical studies evaluated and compared commercial, open-source and research black box SQL injection testing tools [Bau et al., 2010, Doupé et al., 2010, Vieira et al., 2009]. These studies found that black box testing tools have low detection rates and high false positive rates for SQL injections. This result highlights the need to improve both test generation approaches and oracles for SQL injection testing. In this paper, we focus on improving the oracle by using database proxies rather than relying on the output of the application.

Elia et al. [Elia et al., 2010] evaluated several intrusion detection tools, including the database

proxy GreenSQL that we use in this paper. The study injects security faults into the applications under study and then automatically attacks the application to evaluate the effectiveness of the intrusion detection tools studied. These papers focused on testing and comparing security mechanisms, such as WAFs and database proxies, while we propose utilising these tools in the security testing process.

4.5 Summary

SQLi attacks are a major threat to web applications. It is therefore highly important to test such applications in an effective manner to detect SQL injection vulnerabilities. In many situations, for example when the source code is not available or adequate code analysis technologies are not applicable, one must resort to black box testing. This paper examined the impact of WAFs and DIDSs on black box SQL injection testing. We propose to use WAFs to prioritise repairing efforts of SQLi vulnerabilities by testing the application with and without a WAF and then to prioritise fixing vulnerabilities that are not protected by the WAF. We also propose to use DIDSs, which monitor the communications between the application and the database and flag any suspicious SQL statements, as an oracle for SQL injection testing.

We conducted a case study on two service oriented web applications where we compared the effectiveness and efficiency of two SQL injection tools: SqlMap, which is a state-of-the-art black box testing tool with an oracle that analysis the HTTP responses of the tested application, and our prototype tool, which uses a DIDS (GreenSQL) as an oracle. The results confirm that using a DIDS increases the detection rates of SQL injection testing and also results in finding vulnerabilities with significantly lower numbers of test cases. A more detailed investigation of the test cases revealed that using database proxies helps in detecting more vulnerabilities, but a more sophisticated oracle is needed to be able to reason about the vulnerabilities' exploitability, i.e., if an attacker would be able to gain any benefit from the vulnerability.

We also compared the results of the two testing tools when testing through a WAF (ModSecurity) and when testing the applications directly. The results showed that testing through the WAF is more challenging, causing our prototype tool to only detect 50% of vulnerabilities for one application, while SqlMap detected vulnerabilities for neither application. These results have two implications: Firstly, testing through WAFs can be used to prioritise fixing vulnerabilities that are not protected by the WAF. Secondly, the inability of SqlMap to detect any vulnerabilities when testing through the WAF, although some of those vulnerabilities were detectable by our prototype tool, suggests that we need to further improve the test case generation and oracles of black box SQL injection testing. An improved approach for the generation of test cases is introduced in the next chapter. Chapter 8 introduces a test oracle tailored for the needs of SQLi testing.

Chapter 5

μ4*SQL*: An Input Mutation Approach to the Generation of SQL Injection Test Cases

This chapter introduces a black box automated testing approach for SQLi vulnerabilities, called μ4*SQL*. Starting from "legal" initial test cases, the approach applies a set of mutation operators that are specifically designed to increase the likelihood of generating successful SQLi attacks. More specifically, new attack patterns are likely to be generated by applying multiple mutation operators on the same input. Moreover, some of our mutation operators are designed to obfuscate the injected SQL code fragments to bypass security filters, such as WAFs, while others aim to repair SQL syntax errors that might be caused by previous mutations. As a result, our approach can generate test inputs that produce syntactically correct and executable SQL statements that can reveal SQL vulnerabilities, if they exist. By producing SQLi attacks that bypass the firewall and result in executable SQL statements we ensure to find exploitable vulnerabilities as opposed to vulnerabilities that can not be exploited, for example because a filter blocks all attacks. In addition, concrete sample attacks produced by our approach can help developers to fix the source code or the security filter's configuration. Our approach is fully automated and supported by a tool called Xavier.

We have evaluated our approach on some open-source systems that expose web service interfaces. Compared to a baseline approach, called *Std*, which consists of an up-to-date set of 137 known SQLi attack patterns, our approach is faster and is significantly more likely to detect vulnerabilities within a limited time budget. Moreover, when the subject systems are protected by a WAF, **none** of the inputs generated by *Std* that reveal vulnerabilities can get through the firewall, while our approach can still generate a good amount of inputs, getting through the firewall and revealing all-but-one known vulnerabilities.

The remainder of this chapter is organised as follows: Section 5.1 presents our proposed mutation operators and security testing approach. Section 5.2 introduces the implementation of our approach. Finally, Section 5.3 presents the evaluation together with a discussion of results and threats to validity.

5.1 Approach

We propose an automated technique, namely μ4*SQL*, for detecting SQLi vulnerabilities. Our technique rests on a set of mutation operators that manipulate inputs (legitimate ones) to create new test

inputs to trigger SQLi attacks. Moreover, these operators can be combined in different ways and multiple operators can be applied to the same input. This makes it possible to generate inputs that contain new attack patterns, thus increasing the likelihood of detecting vulnerabilities.

Specifically, we want to generate test inputs that can bypass web application firewalls and result in executable SQL statements. A WAF may block SQLi attacks and prevent a vulnerable web service from being exploited. Therefore, effective test inputs need to get through the WAF in order to reach the service. Furthermore, they should lead to executable SQL statements as otherwise, security problems are unlikely to arise since the database engine will reject them and consequently no data would be leaked or compromised.

This section introduces our proposed mutation operators to generate test data. For each mutation operator, along with its definition, a concrete example is provided. In some operators we discuss also their preconditions with respect to input and previously applied operators. We will then discuss our test generation technique and the automated tool we developed to support the technique.

5.1.1 Mutation Operators

Mutation operators (MO) can be classified by their purpose into the following three classes: *Behaviour-changing*, *syntax-repairing* and *obfuscation*. Table 5.1 provides a summary of all mutation operators.

5.1.1.1 Behaviour-Changing

This class of mutation operators mutates inputs with the aim of changing the application's expected behaviour if the application is vulnerable to SQLi. For example, a mutated input could cause the application to return more database rows than expected, exposing sensitive data to an unauthorised user. We define the following behaviour-changing operators:

Example: from original input: *1*; **MO_or** produces a mutated input: *1 OR 1=1*. As a result, if the SQL statement that takes the input is predefined as *"SELECT * FROM table WHERE id="* + *input*, the input will change the logic of the statement and turns it as follows: *SELECT * FROM table WHERE id=1 OR 1=1*. This resulting statement will return all the data of *table*.

Operator: MO_and

Adds *AND x=y* to the *WHERE* clause of an SQL statement where *x* and *y* are random numbers or single characters enclosed in single or double quotes and *x* is not equal to *y*.

Rationale:

By adding a contradiction to the *WHERE* clause, no rows will be affected by the SQL statement. This type of malicious input cannot be used to exploit a vulnerability but because the result of such input is known, this type of input can be used to confirm that a vulnerability is present.

Preconditions:

MO_or has not been applied.

Example: original input: *1*, mutated input: *1 AND 1=2*. That will turn, for example, a predefined statement: *"SELECT * FROM table WHERE id="* + *input* to: *SELECT * FROM table WHERE id=1 AND 1=2*, thus, negating the logic of the original statement.

Table 5.1. Summary of mutation operators classified into behaviour-changing, syntax-repairing, and obfuscation operators.

MO name	Description
Behaviour-Changing Operators	
MO_or	Adds an OR-clause to the input
MO_and	Adds an AND-clause to the input
MO_semi	Adds a semicolon followed by an additional SQL statement
Syntax-Repairing Operators	
MO_par	Appends a parenthesis to a valid input
MO_cmt	Adds a comment command (-- or #) to an input
MO_qot	Adds a single or double quote to an input
Obfuscation Operators	
MO_wsp	Changes the encoding of whitespaces
MO_chr	Changes the encoding of a character literal enclosed in quotes
MO_html	Changes the encoding of an input to HTML entity encoding
MO_per	Changes the encoding of an input to percentage encoding
MO_bool	Rewrites a boolean expression while preserving it's truth value
MO_keyw	Obfuscates SQL keywords by randomising the capitalisation and inserting comments

Operator: MO_uni

Adds the SQL *UNION* operator followed by an additional *SELECT* statement to the input.

Rationale:

The *UNION* operator combines the result set of two or more *SELECT* statements into a single result set. Injecting a *UNION* in an SQL statement allows the attacker to access any table in the database and not limit him/her to tables used in the predefined statement. The result set returned from each *SELECT* statement must have the same number of columns and the i^{th} column in both result sets must be of the same data type, otherwise a run-time error will occur.

Preconditions:

The predefined SQL statement has been determined to be a *SELECT* statement.

Example: original input: *1*, mutated input: *1 UNION SELECT login, password from table2*. This changes the predefined statement: *"SELECT productName, manufacture FROM table WHERE id="* + *input* to *SELECT productName, manufacture FROM table WHERE id=1 UNION SELECT login, password fromtable2*.

Operator: MO_semi

Adds a semicolon (;) followed by an additional SQL statement to the input. The resulting query has the form *sql_stmt1; sql_stmt2*, where *sql_stmt1* is the original SQL statement and *sql_stmt2* is a randomly chosen SQL statement from a predefined list.

Rationale:

SQL statements separated by a semicolon are executed by the server from left to right, unless an error is encountered. If this operator is applied successfully, any SQL statement can be injected after the semicolon giving the attacker complete control over the database if no other restrictions are applied (e.g., the database user has no privilege restrictions defined on the database level).

Example: original input: *1*, mutated input: *1; SELECT waitfor(5) FROM dual*. This changes the predefined statement: *"SELECT * FROM users WHERE id=" + input* to: *SELECT * FROM users WHERE id=1; SELECT waitfor(5) FROM dual*.

5.1.1.2 Syntax-Repairing

As mentioned before, a SQLi attack aims to change the behaviour of the application by injecting malicious inputs. Therefore, the malicious input itself is expected to contain SQL statement fragments. This type of input, unlike regular valid inputs, could cause an SQL syntax error when being combined with its targets, i.e., predefined SQL statements. Since the approach we propose is a black box technique, the predefined SQL statement syntax is unknown to the test generator making it challenging to generate inputs that do not cause syntax errors. This class of mutation operators mutates inputs with the goal of trying to repair SQL syntax errors when they might be encountered. The mutation operators we define in this class are the following:

Operator: MO_par

Appends a closing parenthesis to the end of an input.

Rationale:

In some cases, an input provided by the user is used as a parameter for an SQL function call or within a nested *SELECT* statement. In such scenarios the input is inserted within parenthesis, for example, *func_name(input)*. If such input is vulnerable to injections, this vulnerability can only be exploited when the opening parenthesis of the function call is matched with a closing parenthesis. Otherwise, the injected input would be interpreted as part of the function's parameter, which might cause a syntax error or cause the injection to have no effect on the application's behaviour.

Preconditions:

A behaviour-changing mutation operator has been previously applied.

Example: original input: *67*, mutated input: *67)*. When the input is further mutated with **MO_or** and **MO_cmt**, the obtained mutated input will be: *67) OR 1=1 -{}-*. Let us consider a predefined statement: *"SELECT * FROM table WHERE character=CHR(" + input + ")"*, where function *CHR* converts an integer to its corresponding Unicode character. The changed SQL statement: *SELECT * FROM table WHERE character=CHR(67) OR 1=1 –)*.

Operator: MO_cmt

Adds an SQL comment command (double dashes -- and the hash character #) to the input. Any SQL that follows a comment command is not executed.

Rationale:

an SQL comment command can be useful to repair syntax errors that were caused by previous mutations. By appending an SQL comment command at the end of the mutant, everything following the comment will be ignored by the parser, which might help fixing SQL syntax errors.

Preconditions:

Another operator, such as MO_par, has been previously applied and caused a syntax error.

Example: original input: *67*, after being mutated with **MO_or** and **MO_par**: *67) OR 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE character=CHR(" + input + ")"* to a combined statement, which causes a syntax error: *SELECT * FROM table WHERE character=CHR(67) OR 1=1)*.

We then apply **MO_cmt** to obtain: *67) OR 1=1 #*. The final statement: *SELECT * FROM table WHERE character=CHR(67) OR 1=1 #)* Applying this mutation causes the last parenthesis to be ignored by the parser, thereby avoiding parser error due to the unbalanced number of parentheses.

Operator: MO_qot

Adds either a single quote (') or a double quote (") to the mutant.

Rationale:

If an input that is vulnerable to injections is of type string, it may be enclosed in single or double quotes in the predefined SQL statement. The SQL parser will treat the mutant, including the injected SQL, as a string literal and will not execute any SQL commands within the mutant. To be able to exploit the vulnerability, we have to first exit the string context by closing any open quotes before any SQL commands can be injected.

Preconditions:

A behaviour-changing mutation operator, which contains a character literal, has been previously applied.

Example: original input: Smith, mutated with **MO_or**: *Smith OR 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE name='" + input + "'"* to combined statement, which does not result in the desired change of behavior, since the mutant is treated as a string literal: *SELECT * FROM table WHERE name='Smith OR 1=1')*. After being further mutated with **MO_qot** and **MO_cmt**: *Smith' OR 1=1 #*, the final statement is *SELECT * FROM table WHERE name='Smith' OR 1=1 #)*, which is syntactically correct and changes the logic of the original statement.

5.1.1.3 Obfuscation

Some applications employ input filters, e.g., a web application firewall, to defend against SQLi attacks. In essence, a WAF examines every input to check for suspicious string patterns typically used in SQLi attacks, such as SQL keywords, and blocks them. For example, a WAF uses a blacklist that defines forbidden characters or strings to decide if an input is suspicious. In practice, such filters protect many security-critical systems. For example, a software system, which handles credit card data, has to employ a WAF to prevent attacks and to be compliant with industry security standards. Obfus-

cation mutation operators try to avoid filtering by mutating an input to a semantically equivalent input but in a different form. This might prevent the filter from recognising the forbidden characters/strings in the mutated input. We define the following obfuscation mutation operators:

Operator: MO_wsp

Replaces a whitespace with a semantically equivalent character (+, /**/, or unicode encodings: %20, %09, %0a, %0b, %0c, %0d and %a0).

Rationale:

An application might filter inputs that contain string patterns known to be used in SQLi attacks, for example, a single quote followed by a whitespace. Representing the whitespace in a different encoding might cause the malicious input to avoid this filter.

Preconditions:

The input contains at least one whitespace.

Example: original input: *1 OR 1=1*, mutated input: *1+OR+1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE id=" + input* to *SELECT * FROM table WHERE id=1+OR+1=1*.

Operator: MO_chr

Replaces a character literal enclosed in quotes (*'c'*) with an equivalent representation, where c is an arbitrary printable ASCII character. Equivalent representations are:

- Short binary representation, for example, *'a'* is replaced with *b'1100001'*.
- Long binary representation, for example, *'a'* is replaced with *_binary'1100001'*.
- Unicode representation, for example, *'a'* is replaced with *n'a'*.
- Hexadecimal representation, for example, *'a'* is replaced with *x'61'*.

Rationale:

If a filter rejects a mutant generated by a behaviour-changing mutation operator which contains a character literal, this operator can be used to obfuscate the mutant. For example, MO_or generates mutants with a tautology, e.g. *OR 'a'='a'*. A filter may may be configured to identify suspicious inputs by checking for a tautology pattern. To avoid this filter, this operator changes the representation of one of the tautology's operands, while preserving the semantic meaning. For example, *OR 'a'='a'* could be mutated to *OR 'a'=x'61'*, where *x'61'* is the hexadecimal representation of *'a'*. This new mutant might not be recognized as a tautology by the filter and, therefore, avoid filtering.

Preconditions:

A behaviour-changing mutation operator, which contains a character literal, has been previously applied.

Example: original input: 1, mutated with **MO_or**: *1 OR 'a'='a'*, further mutated with **MO_chr**: *1 OR 'a'=x'61'*. This changes the predefined statement: *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 OR 'a'=x'61'*.

Operator: MO_html

Changes the encoding of a mutant using HTML entity encoding. In HTML entity encoding, a character can be encoded in two ways: (i) numeric character reference in the form *&#N* where *N* is the character's code position in the used character set in decimal or hexadecimal representation; (ii) Character entity reference [W3C, 2012] in the form *&SymbolicName*. For example, *"* is the encoding for the single quote character (').

Rationale:

Using HTML entity encoding might help evade a filter that is designed to reject a certain character, but does not recognize the same character if received in HTML entity encoding.

Preconditions:

For character entity reference encoding, only characters with symbolic names can be encoded.

Example: original input: 1, mutated with **MO_or**: 1 OR 'a'='a', further mutated with **MO_html**: *1 OR "a" = "a"*. This turns the predefined statement: *"SELECT * FROM table WHERE id="* + *input* to: *SELECT * FROM table WHERE id=1 OR "a" = "a"*.

Operator: MO_per

Changes the encoding of a mutant using percent encoding:*%HH*, where *HH* is a two digit hexadecimal value referring to the character's ASCII code. For example, the single quote character (') is encoded as *%27*.

Rationale:

Using percent-encoding is useful if a filter rejects a certain character, but does not recognize the same character if received in percent encoding.

Example: original input: 1, mutated with **MO_or**: *1 OR 'a'='a'*, further mutated with **MO_per**: *1 OR%20'a'='a'*. This turns the predefined statement: *"SELECT * FROM table WHERE id=" + input* to *SELECT * FROM table WHERE id=1 OR%20'a'='a'*.

Operator: MO_bool

Replaces a boolean expression with an equivalent boolean expression. For example, the boolean expression *1=1* which is used in MO_or could be obfuscated as *not false=!!1*. Both expressions evaluate to true, which maintains the same semantic meaning of the mutant after obfuscation.

Rationale:

A filter might be configured to look for and reject a certain boolean expression which is used as part of a request which is frequently used in SQLi attacks, e. g. a tautology. By obfuscating the boolean expression, the filter might fail to recognise the attack making it possible to perform the attack.

Preconditions:

Can only be applied to input values that contain a boolean expression.

Example: original input: *1*, mutated with **MO_or**: *1 OR 1=1*, further mutated with **MO_bool**: *1 OR not false=!!1*. This turns the predefined statement *"SELECT * FROM table WHERE id=" + input* to: *SELECT * FROM table WHERE id=1 OR not false=!!1*.

Operator: MO_keyw

Obfuscates SQL keywords and operators using different techniques: Randomly changing the case of some letters, adding comments in the middle of a keyword or replacing a keyword with an alternative representation. Most SQL parsers are case insensitive, e.g. the keyword *select*, *SELECT* or *SeLeCt* are all valid. Some parsers accept keywords which contain a comment in the middle of the keyword (e.g. *sel/*comment here*/ect*). Finally, some keywords have alternative forms, e.g. *OR* can also be expressed as ||.

Rationale:

A filter might be configured to reject a request that contains any SQL keyword, since SQL keywords are frequently used in SQLi attack strings. By obfuscating the SQL keywords in an input, the filter may fail to recognise those keywords making the attack successful.

Preconditions:

The input value contains at least one SQL keyword.

Example: original input: *1*, mutated with **MO_or**: *1 OR 1=1*, further mutation with **MO_keyw**: *1 || 1=1*. This changes the predefined statement: *"SELECT * FROM table WHERE id="* + *input* to: *SELECT * FROM table WHERE id=1 || 1=1*.

5.1.2 Test Generation

A single or multiple mutation operators of different types can be applied to a single input parameter to generate desired inputs. The latter case aims at detecting subtle vulnerabilities that can only be triggered with an input generated by combining multiple mutation operators. For example, consider an application that filters inputs by searching for known attack patterns that can be generated using one of the behaviour-changing operators. To form a successful attack, it is necessary to first apply a behaviour-changing operator and then apply one or more obfuscation operators.

Each chain of mutations has to start from a valid test case, which satisfies the input validations of the application under test. Starting from a valid test case ensures that we avoid generating test cases that would be directly rejected by the application due to dependencies between inputs or complex input structures that are unlikely to be generated randomly. Moreover, valid test cases have the benefit of being more likely to satisfy input validations and reach critical parts of the application, such as SQL queries. For example, if an application expects a credit card number together with other inputs, which we wish to mutate, the credit card number has to follow a well-defined format; otherwise the test case would be instantly rejected. With the presented approach, valid test cases from existing functional test suites can be reused or, if such test suites do not exist, valid test cases can be manually created using SoapUI[1] and similar tools.

Algorithm 1 formally defines the test generation algorithm: Starting from a valid test case, each input is mutated a predefined number of times. The function *Apply_MO* (Line 4) randomly applies one or more mutation operator(s) to the current *Input*. The function uses a simple grammar that defines the different legal ways to combine operators and ensures that all preconditions for the applied operators are satisfied. The operation under test is then called with the updated test case TC'. If the oracle flags a vulnerability, all SQL statements that were issued as a result of the call are checked. If the percentage of executable SQL statements (i.e., statements that do not contain a syntax error) is above a predefined threshold P, the input is reported as vulnerable and the test case is saved to help the test engineer in debugging and fixing the vulnerability (Line 5-8). In our experiments we choose $P = 100\%$, meaning that all triggered SQL statements must be executable.

Figure 5.1 shows an example of a SOAP message (a test case) generated by our approach. Here the input values of the parameters *minPrice*, *maxPrice*, and *start* are kept from the original test case, while the input value of the parameter *country* has been mutated to contain a SQLi attack.

5.1.3 Test Oracle

When a malicious input is sent to a target system, it may result in making the system misbehave if successful. In most cases, the manifestation of abnormal behaviours can be observed from the results the target system returns (e.g., web pages showing unintended content) or from the surrounding environment (e.g., crashes, illegal calls to the operating system, or unintended accesses to data). In

[1] http://www.soapui.org

Algorithm 1 Test Generation Algorithm

Input:
TC: A test case: ArrayOf(*Input*)
OP: A web service operation to be tested
Output:
TS: Test Suite for SQLi vulnerabilities
V: Set of vulnerable inputs

```
1:  TS = ∅
2:  for all Input in TC do
3:      while max_tries_not_reached do
4:          TC' = apply_MO(TC,Input)
5:          if call(OP,TC') = VulnrFlagged then
6:              if executable_SQL ≥ P then
7:                  V = V ∪ Input
8:                  TS = TS ∪ TC'
9:              end if
10:         end if
11:     end while
12: end for
13: return TS, V
```

```
<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <urn:getRoomsByRate>
        <minPrice xsi:type="xsd:float">100</minPrice>
        <maxPrice xsi:type="xsd:float">400</maxPrice>
        <country xsi:type="xsd:string">"||not 0--</country>
        <start xsi:type="xsd:integer">1</start>
    </urn:getRoomsByRate>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 5.1. Example of a generated test case, the parameter *country* contains a mutated SQLi attack.

our experiments, because we focus on SQL injections, we deploy a database proxy that intercepts the communication between the target system and its database, to identify if an input is potentially harmful or not. For example, we can use **GreenSQL**[2] for this purpose. A previous study that compared GreenSQL to five similar tools has found it to be the most effective in detecting SQL injection attacks [Elia et al., 2010].

Details of using a database proxy as oracle has been discussed in our previous work [Appelt et al., 2013]. Typically, a database proxy is deployed and trained with normal database accesses. Such training data are the results of regular usage of the systems or the execution of existing functional test suites. Based on the training data, the proxy learns regular patterns of legal SQL statements. Once

[2]http://www.greensql.com

trained, the proxy will continue observing the traffic between the system and its database and raise alarms when identifying suspicious database queries. Each alarm corresponds to one database SQL statement, and one test case can result in multiple SQL statements and thus multiple alarms. To avoid false positives due to incomplete training, manual inspection may be needed to verify that all SQL statements flagged actually point to a vulnerability in the system.

5.2 Implementation

The presented mutation approach has been implemented as a Java tool, called Xavier[3]. It can be used to test SOAP-based web services for SQLi vulnerabilities. Figure 5.2 shows the key components of the tool (*Test generator* and *Monitor*) and how it is used in practice. The test generator takes as inputs the WSDL file of the web service under test and a sample test case for each web service operation that has to be tested. Such a sample test case can be easily generated by professional tools, such as SoapUI, or by existing approaches [Bartolini et al., 2009]. The tool, then, examines the sample test case to find all input parameters for an operation and replaces each parameter, one at a time, with a SQLi attack generated with our mutation approach. The modified test case will be sent to the web service under test (the SUT in the figure). In some settings, there could be a web application firewall (the WAF component) deployed in between the test generator and the SUT. The oracle component (the *DB proxy* component in the figure) observes the interactions between the SUT and its database to detect malicious SQL statements. Finally, the *Monitor* component of Xavier constantly queries the oracle component to know whether generated inputs reveal a SQLi vulnerability.

In Xavier, we integrate GreenSQL to intercept SQL statements. The database proxy uses a learning approach to detect SQLi vulnerabilities. Therefore, it has to be trained in a learning phase to recognise legal SQL statements. In the detection phase, the proxy considers all intercepted statements, which have not been learned previously, as SQLi attacks.

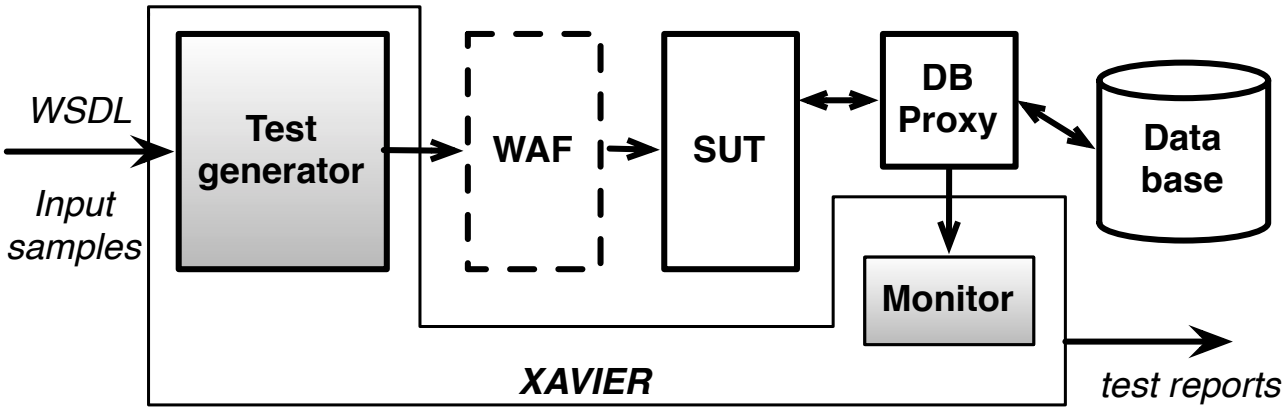


Figure 5.2. Components of Xavier and how Xavier is used in practice.

Every suspected malicious statement is further analysed if it forms syntactically correct SQL. An attacker is only able to exploit a SQLi vulnerability, if he can inject the malicious input in such a way that the resulting SQL statement is free of syntax errors. Otherwise, the attacker is unable to reach his goal, e.g., to obtain/modify data or change the application's control flow, if the malicious statement is

[3]Contact us for download

not executed. The tool MySQL-Proxy[4] is used to monitor if an SQL statement has been executed or if there was an error during execution.

5.3 Evaluation

We have evaluated the effectiveness of our approach on two open source systems and in two different settings: with and without the presence of a web application firewall (WAF). The main motivation for the latter is that, in most contexts, including those of our industry partners, such a firewall is typically present (or sometimes integrated) and is the first protection layer encountered by attackers. Such situations are therefore deemed more realistic. The firewall deployed in our experiment is ModSecurity with the OWASP Core Rule Set (version 2.2.0). As the baseline for our evaluation, we considered a comprehensive list of known SQLi attack patterns.

We aim at evaluating the performance of our proposed mutation technique in comparison with standard attacks. More specifically, we investigate the following research questions:

RQ1: *Are standard attacks and mutated inputs (generated by μ4SQL) likely to reveal exploitable SQLi vulnerabilities?*

RQ2: *With and without the presence of the WAF, which input generation technique performs better?*

5.3.1 Subject Applications

Two open-source subjects, namely HRS and SugarCRM were used in our experiments. HRS was created by researchers to study service-oriented architectures and was used in previous studies [Coffey et al., 2010a]. SugarCRM is a popular customer relationship management system (received 189K+ downloads as of 2013[5]). These systems provide web service APIs to the external world. Such interfaces allow other systems, namely service consumers, to access to the business functionality and data of the subject systems. However, they are also target for SQLi attacks if the inputs through those interfaces are inadequately treated.

Table 5.2. Size in terms of web service operations, parameters, and lines of code of the subject applications.

Application	#Operations	#Parameters	#LoC
HRS	7	21	1,566
SugarCRM	26	87	352,026
Total	33	108	353,592

Table 5.2 provides information about the number of operations, input parameters and lines of code for the chosen applications. In terms of size in number of lines of code, HRS and SugarCRM,

[4]http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html
[5]http://sourceforge.net

with 1.5KLoCs and 352KLoCs respectively, are not particularly large but they provide a respectable number of services with many input parameters, with known vulnerabilities. SugarCRM and HRS are both implemented using PHP, use a MySQL database, and provide a SOAP-based Web Service API. Those are popular technologies used in the implementation of many web services.

5.3.2 Treatments

We refer to the baseline approach consisting of 137 known attack patterns as *Std* (Standard attacks). Such patterns were consolidated in a repository of SQLi attack patterns [Antunes et al., 2009]. They include different contemporary categories of attacks, such as *Boolean-based*, and *UNION query-based*. In the context of our study, the whole set of attack patterns of *Std* is applied for every individual input parameter. The second treatment used in our study is our approach, $\mu 4SQL$.

5.3.3 Variables

We used **GreenSQL** [6] as the oracle (database proxy) for the generated test inputs of *Std* and $\mu 4SQL$. To train GreenSQL, we have created a test suite for each of the subjects consisting of a wide range of legitimate input values. To avoid false positives due to incomplete training, in our experiments, we manually verified that all SQL statements flagged by GreenSQL did actually point to a vulnerability in the system and were not legal statements which had simply not been learned.

Given a set of test cases targeting a specific web service parameter, we define T as the total number of test cases that generate SQL statements that are flagged (alarm) by the database proxy. Among these tests, we further investigate if their generated SQL statements are executable or not. We refer to T_e for the total number of tests that can lead to flagged and executable SQL statements. To compare *Std* and $\mu 4SQL$, we need to consider both T and T_e, as we will see that looking at T alone would lead to very different conclusions since only executable SQL statements can be exploitable. Non-executable statements can be generated because the corresponding inputs, after being processed by a target, result in syntax-errors. Such statements hardly have a security impact since the database engine would reject them and, hence, no data would be leaked or compromised.

If a technique yields higher T_e, it is considered to be more effective at detecting exploitable vulnerabilities. In other words, when T_e is high, it is more likely to detect exploitable vulnerabilities for a test suite of fixed size. Moreover, it is also likely to detect vulnerabilities faster, i.e., we need a smaller number of tests to be executed in order to detect the vulnerabilities. This, in practice, is important when dealing with a large number of services and input parameters.

Since one test case can give rise to multiple SQL statements, we need to determine how to compute T_e when there is a mix of executable and non-executable statements. Since, in practice, one single flagged and executable statement generated by a specific input can entail serious consequences, when more SQL statements are executable, the chance to uncover vulnerabilities is higher. In our analysis, with the intent of being conservative in our results, a test t is considered to be part of T_e if and only if all the flagged statements generated by t are executable.

[6] http://www.greensql.com

5.3.4 Results

We ran μ4*SQL* and *Std* on every parameter of the two selected subjects, SugarCRM and HRS. There are in total 108 input parameters for all their web services. As described earlier, *Std* entails 137 test executions for every parameter, whereas with μ4*SQL*, since it is non-deterministic, we need to run more test executions to account for randomness. To do so in an efficient way, given the substantial execution time (about 5.7 hours per vulnerable parameter on a virtual machine of 1Gb RAM and 2,6Ghz CPU) we generated and ran 1000 tests for each parameter. We, then, adopted a Bootstrapping approach (sampling with replacement) [Efron and Tibshirani, 1993] and formed 10k test suites (each has 137 tests) by sampling from these 1000 tests, so that each test suite would be comparable with *Std* with respect to T_e. In the tables 5.3 and 5.4, we report the percentage of T and T_e for *Std* on each subject and their average percentage for μ4*SQL* over 10k test suites. We only report results for vulnerable input parameters as per the results of the two test techniques and after being confirmed through manual inspection.

Table 5.3. Results of *Std* and μ4*SQL* on the subject applications when no WAF is enabled.

Subject	**Parameter**	*Std*		μ4*SQL*	
		$\%T$	$\%T_e$	$\%T$ (avg)	$\%T_e$ (avg)
HRS	country	12.41	5.84	40.62	21.80
	arrDate	35.04	9.49	42.05	12.50
	depDate	35.04	9.49	42.96	12.03
	name	35.04	9.49	43.36	12.91
	address	35.04	9.49	39.81	11.00
	email	35.04	9.49	41.73	11.24
SugarCRM	value	37.23	0.0	41.48	22.51
	ass_user_id	32.85	8.03	42.49	13.91
	query1	32.85	3.65	9.82	0.30
	query2	54.74	5.84	81.72	33.45
	order_by	59.85	10.95	85.98	33.55
	rel_mod_qry	47.45	2.92	49.79	0.00

Table 5.3 shows our results when subjects were not protected by the WAF. The first and second column indicate the subjects and their vulnerable parameters, the subsequent columns show the percentage of tests that generate flagged SQL statements ($\%T$) and the percentage of such flagged tests (out of 137) that also lead to executable SQL statements ($\%T_e$). For μ4*SQL*, as indicated, such percentages are averages over 10k test suites. For HRS, both techniques find six SQLi vulnerabilities. With regards to the parameter *country*, GreenSQL flags 12.41% of 137 test cases of *Std* as SQLi attacks and, among them, 5.84% generate executable SQL statements. Results for the remaining five parameters found to be vulnerable by *Std* are identical: 35.04% of the tests lead to SQL statements being flagged by GreenSQL and among them, 9.49% generate executable SQL statements. While μ4*SQL* and *Std* detect the same vulnerabilities, $\%T$ and $\%T_e$ are higher for μ4*SQL*: across reported parameters, T ranges from 39.81% to 43.36% and T_e from 11% to 21.80%. For SugarCRM, both techniques detect five out of six vulnerabilities, but both *Std* and μ4*SQL* failed to generate an executable SQL statement for one parameter, that is *value* and *rel_mod_query*, respectively. Except for

parameter *query1* μ4*SQL* has always a higher T measure. Similarly, μ4*SQL* has a higher T_e measure for all parameters except for *query1* and *rel_mod_query*.

Even when using μ4*SQL*, $\%T_e$ is generally lower than $\%T$ across input parameters. However, it is large enough to be highly likely to detect an exploitable vulnerability by running a few dozens test cases or less, as only one flagged test case leading to an executable SQL statement is enough to demonstrate the vulnerability of a parameter. Taking the parameter *ass_user_id* as an example, with a average $\%T_e$ of 13.91%, running 50 test cases would yield a very small probability, 0.0006 (i.e., $(1-0.1391)^{50}$), of missing the vulnerability. $\%T$ is typically much larger than $\%T_e$, thus showing that generating executable SQL statements is rather difficult.

Table 5.4. Results of *Std* and μ4*SQL* on the subject applications protected by the WAF.

Subject	**Parameter**	*Std*		μ4*SQL*	
		$\%T$	$\%T_e$	$\%T$ (avg)	$\%T_e$ (avg)
HRS	country	0.73	0.0	36.84	20.69
	arrDate	2.19	0.0	35.91	9.11
	depDate	5.84	0.0	36.59	11.42
	name	6.57	0.0	38.34	11.72
	address	7.30	0.0	39.67	9.64
	email	6.57	0.0	36.33	9.88
SugarCRM	value	2.19	0.0	37.42	20.48
	ass_user_id	5.11	0.0	29.35	6.89
	query1	0.73	0.0	8.97	0.20
	query2	3.65	0.0	76.56	31.43
	order_by	7.30	0.0	80.08	31.96
	rel_mod_qry	6.57	0.0	44.82	0.0

Table 5.4 shows the results of the experiments when the subjects were protected by the WAF. For HRS, once again, both approaches were able to generate for each vulnerable parameter SQLi statements that were flagged by GreenSQL ($\%T > 0$). However, one important difference is that only μ4*SQL* was able to generate test cases that lead to executable SQL statements. *Std* failed to do so for *all* tested parameters. Similarly, for SugarCRM, $\%T$ is significantly higher for μ4*SQL* than *Std*. And once again, only μ4*SQL* was able to generate test cases that led to executable SQL statements for five out six vulnerable parameters (except *rel_mod_qry*), whereas *Std* failed to do so for *all* tested parameters. Our conclusions are similar to the results when no WAF is present, except that $\%T$ and $\%T_e$ tend to be lower with a WAF. This is to be expected as some of the attacks generated are filtered out by the WAF.

Regarding the performance of μ4*SQL* on the parameter *query1*, the vulnerability, though not impossible to find, is still extremely difficult to detect (only 0.3% of test cases can uncover it). Further work is needed to investigate the reasons.

We further examined why μ4*SQL* experienced, for parameter *rel_mod_qry*, a sharp drop from T (49.72% without WAF) to T_e (0%). μ4*SQL* failed to trigger an executable statement for this parameter since, given the SQL statement into which the test case is injected, non of the mutation operators could possibly result in a syntactically correct statement. The vulnerable statement is:

*SELECT opportunity_id id FROM accounts_opportunities , opportunities WHERE [...] AND <**test case inserted here**> AND [...]*

The injection occurs in the *where* clause of the SQL statement. **MO_or** and **MO_and** are the mutation operators that target SQLi vulnerabilities in the where clause. For both of these operators, all generated mutants for this particular SQL statement begin either with a single quote or a double quote, e.g. *"||'d'='d'–* or *' or 1*, but since there is no matching opening single or double quote an syntax error is introduced. For example, once concatenated with the mutant the statement becomes:

SELECT opportunity_id id FROM accounts_opportunities , opportunities WHERE [...] AND "||'d'='d'– AND [...]

Improving how the mutation operators append a clause can solve this problem. For example, in this particular case, starting the mutant with a number instead of a quote prevents a syntax error. With this additional fix, the vulnerability will be detected. More generally, we expect that the performance of µ4*SQL* will be further improved once we improve the mutation operators.

Answering the research question **RQ1**, we can see that both techniques can, in most cases, reveal vulnerabilities ($\%T_e > 0$) when the subjects were not protected by the WAF. However, when they are protected, only µ4*SQL* can reveal such vulnerabilities (in 10 out of 12 parameters) while *Std* revealed none of them. Such a difference is highly significant as it has many practical implications to be discussed below.

> ***RQ1**: Both the mutation-based technique and the standard attack patterns can reveal SQLi vulnerabilities when no firewall was used. Most vulnerabilities are highly likely to be detected with at most a few dozen test cases or less.*

To provide a better view of the comparison between the two input generation techniques, we produced a set of plots. All of them are available in the appendix. Figures 5.3 and 5.4 depict the results when the subjects were protected with a WAF. The box-plots depict the results of µ4*SQL* (recall it is non-deterministic) in terms of $\%T$ (lower part) and $\%T_e$ (upper part). The dash and triangle dots are the result of *Std*. As we can see, without having to resort to a statistical test, the differences are clearly significant. In the upper part of the figures, none of the tests generated by *Std* could result in executable SQL statements and therefore missed all the vulnerabilities. By contrast, µ4*SQL* missed only one of the vulnerabilities in SugarCRM. In short, from the figures and above tables, we can see that the performance of µ4*SQL* in terms of generating tests that lead to flagged and executable SQL statements is significantly better than *Std*.

> ***RQ2**: Our mutation-based technique (µ4SQL) generates a higher percentage of tests that can reveal SQLi vulnerabilities. Further, in the presence of a WAF, µ4SQL is the only technique that is capable of doing so.*

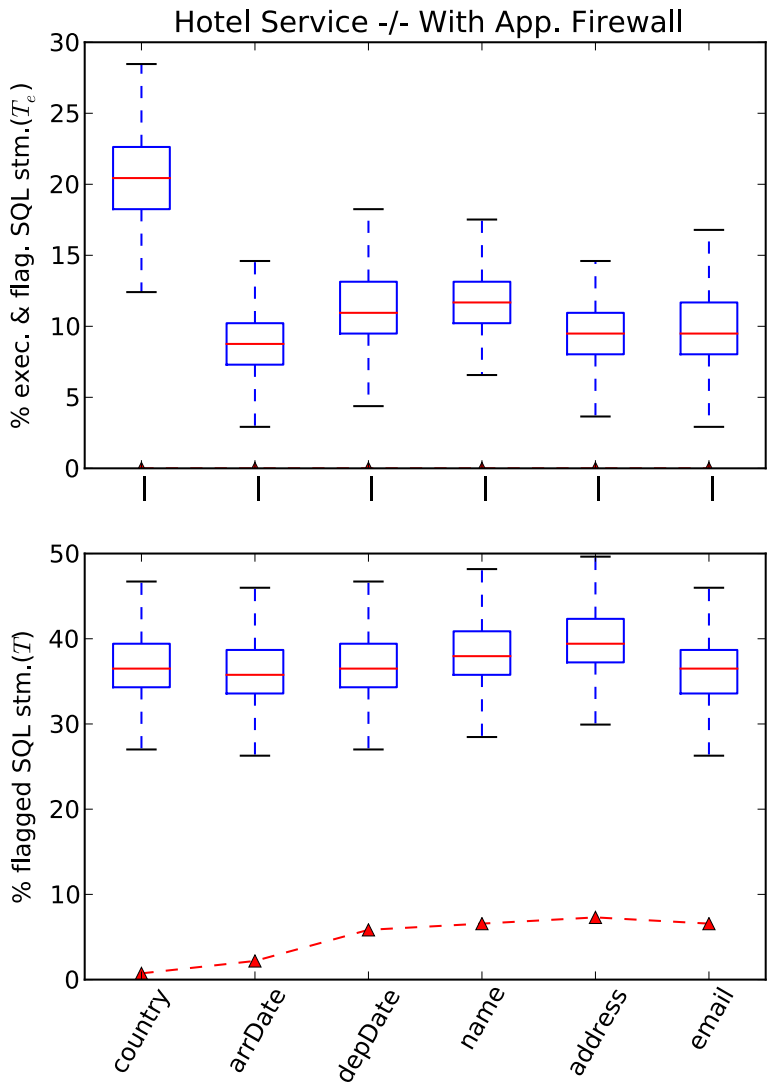


Figure 5.3. Results obtained from HRS with firewall enabled: the box-plots depict the results of μ4*SQL*, the dashed line depicts the results of *Std*. None of the executable SQL statements generated by *Std* can get through the WAF.

5.3.5 Discussion

Results without the WAF indicate that both approaches can detect vulnerabilities in the examined subjects. Both techniques were able to provide, for most vulnerable parameters, test cases leading to SQL statements that are flagged by GreenSQL and deemed executable. It is interesting to note, however, that a significantly higher percentage of test cases generated flagged and executable statements when using μ4*SQL*. The practical implications of these results is that, since the execution time of a test case generated by either *Std* or μ4*SQL* is comparable, when testing many services with many input parameters, μ4*SQL* will be a more effective and less costly technique to detect exploitable vulnerabilities. They will be more likely to be detected within a fixed test budget and will be detected faster.

Results with the WAF are even more dramatic. Only μ4*SQL* is able, for all parameters but one, to generate flagged and executable SQL statements. Since the presence of a WAF or similar protection

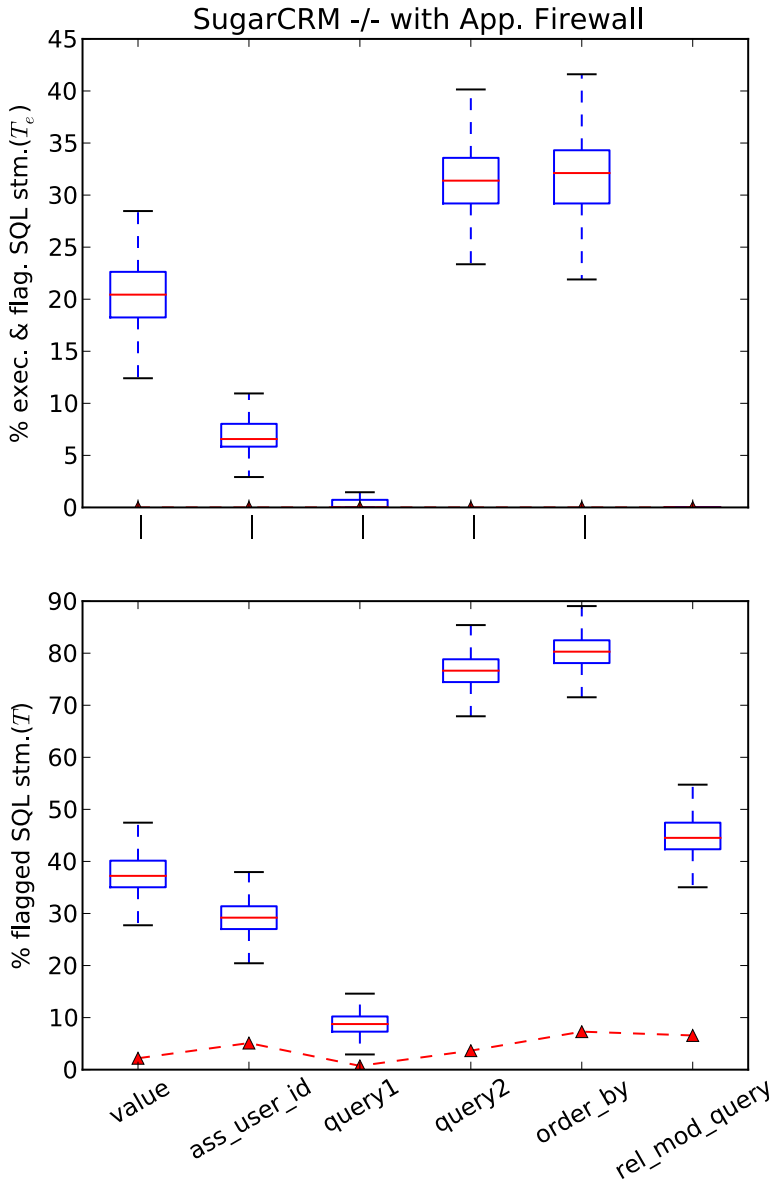


Figure 5.4. Results obtained from SugarCRM with the firewall enabled.

mechanism is a much more realistic situation in practice, these results imply that in many situations, standard attacks are not effective when looking for tangible evidence that there are *exploitable* SQLi vulnerabilities.

When the subjects were protected by the WAF, there was an even more contrasting difference in the results of the techniques. With the WAF enabled, $\mu 4SQL$ achieved results that are similar to when no WAF was used: T and T_e experienced only a slight drop. That difference was due to the WAF identifying and blocking only a small number of attacks. This is an evidence that the proposed obfuscation mutation operators are effective at bypassing the WAF. On the contrary, the test results for *Std* dropped considerably. T experienced a large drop and T_e went down to zero. This can be attributed to the WAF recognising most of the test cases as SQLi attacks and blocking them. The low percentage of test cases which bypass the firewall do not result in executable SQL statements.

Overall, the results indicate that the obfuscation and syntax-repairing have helped $\mu 4SQL$ in bypassing the WAF and triggering executable SQLi attacks.

5.3.6 Threats to Validity

The potential threats to validity of our results fall into the internal and external categories:

Internal threats: This is about whether the associations we observed between treatments (test techniques) and generated executable SQL statements can be confidently interpreted as due to the inherent properties of the techniques. For *Std* we used a comprehensive list of 137 known attack patterns mentioned in [Antunes et al., 2009]. As far as we are aware, this is the state of the art for penetration testing in practice. Regarding $\mu 4SQL$, since it is non-deterministic and to account for randomness, we generated and ran 1000 tests per parameter and then sampled (with replacement, a procedure called Bootstrapping) 10K test suites of 137 test cases to enable a statistical comparison with standard attacks. We have also inspected the reports of GreenSQL to remove any false alarms.

We chose ModSecurity as a WAF and used the OWASP Core Rule Set. This is a popular setting in practice and used in many production systems.

External threats: This concerns the generalization of the results. Obviously, like any study in specific systems, it needs to be replicated. The computation cost of running such experiments is however high and, although we only used two systems in the experiments, them are from different domains and SugarCRM is used by real users as the number of downloads indicates. Although we compared only two test techniques, they are representative of the state of the art in black box SQLi testing, as the review of related works indicates.

5.4 Related Work

Previous research on SQLi detection used both white-box and black box approaches to detect vulnerabilities. Several white-box approaches used taint analysis to identify invalidated inputs that flow into SQL statements [Kieyzun et al., 2009b, Shin, 2006, Smith et al., 2010, Wassermann and Su, 2007]. Fu and Qian [Fu and Qian, 2008] suggested using symbolic execution to identify the constraints that need to be satisfied to lead to a SQLi attack. Shar et al. [Shar et al., 2013] used data mining of the

source code to predict vulnerabilities. As well as requiring access to the source code, which as we mentioned before might not always be possible, most of these approaches rely, in some aspects of their algorithms, on a set of known vulnerability patterns.

Existing black box approaches also rely on known injection patterns when generating test cases. Ciampa et al. [Ciampa et al., 2010] proposed an approach that analyses the output, including error messages, of both legal and malicious test cases to learn more about the type and structure of the back-end database. This information is then used to craft attack inputs that are more likely to be successful at revealing vulnerabilities. Antunes et al. [Antunes et al., 2009, Antunes and Vieira, 2009] also analysed the difference in the behaviour of an application when using malicious and legal inputs to detect vulnerabilities. Huang et al. [Huang et al., 2003] used a test generation approach that uses known attack patterns.

Known SQLi patterns have been enumerated and discussed by various academics [Antunes et al., 2009, Antunes and Vieira, 2009, Halfond et al., 2006b] and online security sources [The Open Web Application Security Project (OWASP), 2013, SQL Injection Wiki, 2013]. However, relying on these patterns might not be sufficient to test an application as attackers are always finding new techniques to exploit vulnerabilities. Moreover, there might be a large number of different representations for the same pattern, for example, using different encodings.

Some approaches proposed run-time prevention techniques rather than testing techniques. In the majority of these approaches [Halfond and Orso, 2005b, Halfond and Orso, 2006, Lee et al., 2012, Shin et al., 2006, Wei et al., 2006], static analysis is used to collect all possible forms of SQL statements that can be produced by the program. At run-time, if the structure of an SQL statement does not match any of those collected forms, the statement is flagged as a potential attack. Sekar [Sekar, 2009] combined taint analysis and policies to detect injection attacks at run-time. Run-time prevention approaches are complementary to testing approaches and can also be used as an effective oracle for testing [Appelt et al., 2013].

Mutation testing has been proposed and studied extensively [Jia and Harman, 2011] as a method of evaluating the adequacy of test suites where the program under test is mutated to simulate faults. Shahriar and Zulkernine [Shahriar and Zulkernine, 2008] defined SQLi specific mutation operators to evaluate the effectiveness of a test suite in finding SQLi vulnerabilities. Mutation analysis was also used by Fonseca et al. [Fonseca et al., 2007] to compare the effectiveness of commercial security testing tools. The mutation operators we propose in this paper mutate test inputs to increase the likelihood of triggering vulnerabilities rather than the program under test to evaluate the effectiveness of test suites in finding faults.

Holler et al. [Holler et al., 2012] proposed an approach called LangFuzz to test interpreters for security vulnerabilities, such as memory safety issues, by mutating the input code. The approach has been applied successfully to uncover defects in Mozilla JavaScript and PHP interpreter. However, our approach differs in various aspects: (1) We target SQL injection vulnerabilities that require different mutation operators and test generation techniques; (2) The observability of failures in the case of SQL vulnerabilities is much more challenging than looking for crashes. We need to intercept communication between a SUT and its database to analyse SQL statements for executability and vulnerability detection.

5.5 Summary

SQL injections have been ranked among the most common and dangerous category of web vulnerabilities. Security testing can be applied to find such vulnerabilities and prevent their exploitation. Automated testing techniques are important, not only to detect vulnerabilities in web services before they can be published, but also to reduce testing effort in contexts where the numbers of services and their input parameters are large. In particular, there is a need for black box techniques that do not require access to the source code, as this is a common constraint when third party components are used or software development is (partly) outsourced. Existing techniques that have investigated this specific problem are bound to known attack patterns that may become out-dated, especially given the fast evolution of web services and the technologies they are build on. Their performance may also be limited by the presence of application protection mechanisms, such as WAFs, which may block known attacks. Our results confirm this problem by showing that state-of-practice, standard attacks do not, in most cases, make it through the firewall. In addition, the few that were not blocked by the firewall lead to non-executable SQL statements because of syntax errors.

We presented in this paper an automated mutation technique for SQL injection vulnerabilities, supported by a tool, which focuses on mutating the input values of web service parameters. This technique makes use of a set of mutation operators that are able (1) to generate inputs with a high likelihood of modifying the behaviour of services, (2) to correct inputs to remove possible syntax errors due to mutations, and (3) to obfuscate attacks to increase their chances to make it through the firewall. The ultimate goal of our technique is to generate randomised inputs to detect SQLi vulnerabilities that lead to executable SQL statements, are passing the firewall, and are unduly revealing or compromising data in the database. Our experimental results have demonstrated that our technique and tool performed much better than state-of-practice standard attack patterns, and that the probability of detecting SQL injection vulnerabilities is high, even in the presence of a firewall, and with a reasonable number of test case executions for each input parameter in each service.

Chapter 6

A Machine Learning-Driven Approach to Testing Web Application Firewalls

WAFs are an indispensable mechanism to protect online systems from attacks. However, the fast pace at which new kinds of attacks appear and their increasing sophistication require WAFs to be updated and tested regularly as otherwise they will be circumvented. In this chapter, we focus our research on WAFs and SQL injection attacks, but the general principles and strategy could be adapted to other contexts. We present a machine learning-driven testing approach to automatically detect holes in WAFs that let SQLs injection attacks bypass them. At the beginning, the approach can automatically generate diverse attacks and then submit them to a system that is protected by a WAF. Incrementally learning from the attacks that are blocked or bypassing the WAF, our approach can then select attacks that exhibit characteristics associated with bypassing the WAF and mutate them to efficiently generate new bypassing attacks. We developed a tool that implements the approach and evaluated it on ModSecurity, a widely used WAF, and a proprietary WAF that protects a financial institution. Evaluation results indicate that our proposed technique is efficient at generating SQL injection attacks that can bypass a WAF and can be used to identify successful attack patterns.

The key contributions in this chapter are:

- A machine learning-driven testing technique for WAFs with an adaptive test selection and generation heuristic, which effectively generates attacks that are likely to bypass a tested WAF, Section 6.1.3.
- A large-scale evaluation with two WAFs, an open-source WAF and a proprietary WAF, that protects a financial institution, Section 6.2.5.
- Assessing the influence of the selected machine learning algorithm on the test results by comparing two alternative classification models, both adapted to large numbers of features and datasets, but with complementary advantages and drawbacks: RandomTree and RandomForest, Section 6.1.3.3 and Section 6.2.5.2.
- An analysis how the found bypassing attacks and other testing output can be used to improve a WAF's rule set and to increase its the attack detection capabilities, Section 6.2.5.4.

Overall, three variants of our machine learning-driven and a random test strategy, which serves as a baseline, are evaluated on two popular WAFs that protect three open-source and 44 proprietary web services. In total, 84 web service parameters are tested, 10 times each for every technique. These

experiments were conducted on a high performance computing cluster and the total computation time is equivalent to 11.5 years on a single CPU core (3.5 weeks of parallel execution) [Varrette et al., 2014].

The obtained results show that our proposed technique performs better compared to the baseline. It is effective in generating many distinct attacks that are not correctly identified by the tested WAFs. Furthermore, our approach enables the identification of attack patterns that are strongly associated with bypassing the WAFs, thus providing support for improving their rule set. All proposed techniques are implemented in an automated testing tool called Xavier.

The remainder of this chapter is structured as follows. Section 6.1 discusses in detail our approach, followed by Section 6.2, where we describe our experiments and results. Besides, we include two appendixes. The first one reports on an in-depth investigation of the differences between the different variants of our proposed machine learning-driven techniques, thus providing a rationale for the testing technique proposed in this chapter; the second one reports detailed results for the conducted large-scale experiment.

6.1 Approach

This section introduces our proposed approach for testing WAFs. Section 6.1.1 defines the input space for this testing problem in form of a context-free grammar. Section 6.1.2 presents a simple attack generation strategy that randomly samples the input space and serves as baseline. Section 6.1.3 presents a test generation strategy that utilises machine learning to guide the test generation towards areas in the input space that are more likely to contain successful attacks. Finally, Section 6.1.4 details the improvements to our proposed attack generation strategy compared to our previous work [Appelt et al., 2015].

6.1.1 A Context-Free Grammar for SQLi Attacks

SQLi attack strings (or tests in our context) are small "programs" that aim at changing the intent of a target SQL statement when they are injected into the statement. Therefore, we systematically surveyed existing SQLi attacks published in the literature, e.g., [Halfond et al., 2006a, Antunes et al., 2009, Antunes et al.,] and from other sources e.g., OWASP[1], and SqlMap[2]. We then defined a context-free grammar for SQLi attacks for generating and analysing SQLi attacks.

The grammar is defined in the Extended Backus Normal Form. An excerpt of the grammar is listed as follows, in which <*start*> is the start symbol, "::=" is the production symbol, "," is concatenation, and "|" represents alternatives.

⟨*start*⟩ ::= ⟨*numericContext*⟩ | ⟨*sQuoteContext*⟩
| ⟨*dQuoteContext*⟩ ;

---

[1] https://www.owasp.org
[2] http://sqlmap.org

⟨*numericContext*⟩ ::= ⟨*digitZero*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩
| ⟨*digitZero*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*digitZero*⟩
| ⟨*digitZero*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sQuoteContext*⟩ ::= ⟨*squote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*squote*⟩
| ⟨*squote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*squote*⟩
| ⟨*squote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*dQuoteContext*⟩ ::= ⟨*dquote*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*dquote*⟩
| ⟨*dquote*⟩, ⟨*parC*⟩, ⟨*wsp*⟩, ⟨*booleanAttack*⟩, ⟨*wsp*⟩, ⟨*opOr*⟩, ⟨*parO*⟩, ⟨*dquote*⟩
| ⟨*dquote*⟩, [⟨*parC*⟩], ⟨*wsp*⟩, ⟨*sqliAttack*⟩, ⟨*cmt*⟩ ;

⟨*sqliAttack*⟩ ::= ⟨*unionAttack*⟩ | ⟨*piggyAttack*⟩ | ⟨*booleanAttack*⟩ ;

⟨*unionAttack*⟩ ::= ⟨*union*⟩, ⟨*wsp*⟩, [⟨*unionPostfix*⟩], ⟨*opSel*⟩, ⟨*wsp*⟩, ⟨*cols*⟩
| ⟨*union*⟩, ⟨*wsp*⟩, [⟨*unionPostfix*⟩] , ⟨*parO*⟩, ⟨*opSel*⟩, ⟨*wsp*⟩, ⟨*cols*⟩,⟨*parC*⟩ ; ⟨*union*⟩ ::= ⟨*opUni*⟩ | `/*!`, `[50000]`, ⟨*opUni*⟩, `*/` ; ⟨*unionPostfix*⟩ ::= `all`, ⟨*wsp*⟩ | `distinct`, ⟨*wsp*⟩; ⟨*cols*⟩ ::= ⟨*digitZero*⟩; ⟨*piggyAttack*⟩ ::= ⟨*opSem*⟩, ⟨*opSel*⟩, ⟨*wsp*⟩, ⟨*funcSleep*⟩;

⟨*booleanAttack*⟩ ::= ⟨*orAttack*⟩ | ⟨*andAttack*⟩ ;

⟨*orAttack*⟩ ::= ⟨*opOr*⟩, ⟨*booleanTrueExpr*⟩ ; ⟨*andAttack*⟩ ::= ⟨*opAnd*⟩, ⟨*booleanFalseExpr*⟩ ;

⟨*booleanTrueExpr*⟩ ::= ⟨*unaryTrue*⟩ | ⟨*binaryTrue*⟩ ;

⟨*binaryTrue*⟩ ::= ... ; ⟨*binaryTrue*⟩ ::= ⟨*unaryTrue*⟩, ⟨*opEqual*⟩, ⟨*wsp*⟩, ⟨*parO*⟩, ⟨*unaryTrue*⟩, ⟨*parC*⟩
| ⟨*unaryFalse*⟩, ⟨*opEqual*⟩, ⟨*wsp*⟩, ⟨*parO*⟩, ⟨*unaryFalse*⟩, ⟨*parC*⟩
| ⟨*squote*⟩, ⟨*char*⟩, ⟨*squote*⟩, ⟨*opEqual*⟩, ⟨*squote*⟩, ⟨*char*⟩, ⟨*squote*⟩
| ⟨*dquote*⟩, ⟨*char*⟩, ⟨*dquote*⟩, ⟨*opEqual*⟩, ⟨*dquote*⟩, ⟨*char*⟩, ⟨*dquote*⟩
| ⟨*unaryFalse*⟩, ⟨*opLt*⟩, ⟨*parO*⟩, ⟨*unaryTrue*⟩, ⟨*parC*⟩
| ⟨*unaryTrue*⟩, ⟨*opGt*⟩, ⟨*parO*⟩, ⟨*unaryFalse*⟩, ⟨*parC*⟩
| ⟨*wsp*⟩, ⟨*trueAtom*⟩, ⟨*wsp*⟩, ⟨*opLike*⟩, ⟨*wsp*⟩, ⟨*trueAtom*⟩
| ⟨*unaryTrue*⟩, ⟨*wsp*⟩, ⟨*opIs*⟩, ⟨*wsp*⟩, ⟨*trueConst*⟩
| ⟨*unaryFalse*⟩, ⟨*wsp*⟩, ⟨*opIs*⟩, ⟨*wsp*⟩, ⟨*falseConst*⟩
| ⟨*unaryTrue*⟩, ⟨*opMinus*⟩, ⟨*parO*⟩, ⟨*unaryFalse*⟩, ⟨*parC*⟩ ;

⟨*terOne*⟩ ::= `1` ; ⟨*squote*⟩ ::= `'` ; ⟨*dquote*⟩ ::= `"` ; ⟨*digitZero*⟩ ::= `0` ; ⟨*digitOne*⟩ ::= `1` ; ⟨*digitXZero*⟩ ::= ⟨*digitOne*⟩ | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` ; ⟨*digitIncZero*⟩ ::= ⟨*digitZero*⟩ | ⟨*digitXZero*⟩ ; ⟨*char*⟩ ::= `a`;

⟨*opNot*⟩ ::= `!` | `not` ; ⟨*opBinInvert*⟩ ::= ~ ; ⟨*opEqual*⟩ ::= = ; ⟨*opLt*⟩ ::= < ; ⟨*opGt*⟩ ::= > ; ⟨*opLike*⟩ ::= `like` ; ⟨*opIs*⟩ ::= `is` ; ⟨*opMinus*⟩ ::= − ; ⟨*opOr*⟩ ::= `or` | `||` ; ⟨*opAnd*⟩ ::= `and` | `&&` ; ⟨*opSel*⟩ ::= `select` ⟨*opUni*⟩ ::= `union` ; ⟨*opSem*⟩ ::= `;` ;

⟨*cmt*⟩ ::= `#` | ⟨*ddash*⟩, ⟨*blank*⟩; ⟨*ddash*⟩ ::= −

⟨*inlineCmt*⟩ ::= `/**/` ; ⟨*blank*⟩ ::= ␣ ; ⟨*wsp*⟩ ::= ⟨*blank*⟩ | ⟨*inlineCmt*⟩

This grammar can be extended to incorporate other variants of SQLi attacks. For example, the non-terminal <*blank*> can have more semantically equivalent terminal characters: +, /**/, or unicode encodings: %20, %09, %0a, %0b, %0c, %0d and %a0; the quotes (single or double) can be represented using HTML encoding, and so on.

6.1.2 Grammar-based Random Attack Generation

Based on the proposed grammar, the random attack generation (RAN) procedure is straightforward: beginning from the start symbol, a randomly selected production rule is applied recursively until only terminals are left. Since there is no loop in the grammar, the attack generation will always terminate. The output SQLi attack is produced by concatenating all terminal symbols.

In order to produce a set of diverse random SQLi attacks that yield a good coverage of the grammar, each production rule is selected with a probability proportional to the number of distinct descendant production rules from the current one.

Amongst the techniques presented in this work, RAN implements the most simplistic strategy for sampling the input space defined by the attack grammar.

6.1.3 Machine Learning-Guided Attack Generation

If the difficulty to find bypassing attacks increases, i.e. because a WAF detects a large share of attacks, a random attack generation strategy like RAN might prove to be inefficient. In such situations, a more advanced test generation strategy that spends more computational effort on generating smart test cases is expected to be more efficient. In this section we introduce a machine learning based approach, called ML-Driven, to sample the input space in a more efficient manner than RAN does.

ML-Driven is inspired by search-based test generation [Banzhaf et al., 1998, McMinn, 2004, Anand et al., 2013]. We face the problem to efficiently choose from a large set of SQLi attacks the ones that are more likely to reveal holes in the WAF under test. The problem is challenging because there is little information available to estimate how close a test comes to bypassing the WAF. When a test is executed only one of the following two events can be observed: *bypassing*, or *blocked*. This leaves the search with no guidance to effectively assess how close a blocked attack is from bypassing the WAF. To tackle the problem, we use machine learning to model how the elements (attributes of attacks) of the tests are associated with the likelihood of bypassing the WAF. In the search process, tests that are predicted to have such high likelihood are considered to have a high fitness and are more likely to be generated.

More specifically, ML-Driven employs, first, the random test generation technique described in the previous section to generate an initial training set. These tests are sent to a web application protected by the WAF. Depending on whether they bypass or are blocked by the WAF, they are labelled as "P" or "B", respectively. We encode these tests and use them as initial training data to learn a model estimating the likelihood (f) with which tests can bypass the WAF. Using this measure we can rank, select, and modify tests associated with high f values to produce new tests. These new tests are then executed, and their results ("P" or "B") are used to improve the prediction model, which will in turn help generating more distinct tests that bypass the WAF.

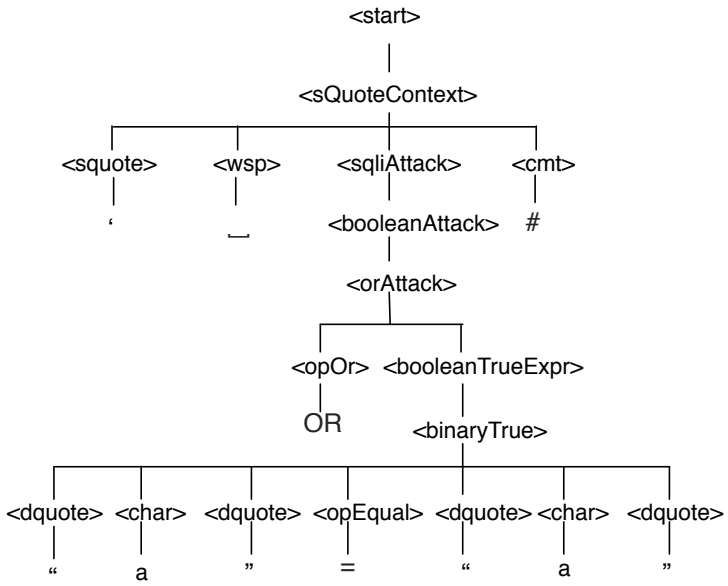


Figure 6.1. The derivation tree of the “boolean” SQLi attack: *’␣OR“a”=“a”#*.

In what follows, we will discuss in detail how tests are decomposed and encoded for machine learning and the mutation process, which is the process to generate new SQLi attacks from previously generated attacks. Finally, we describe ML-Driven , a machine learning test generation approach.

6.1.3.1 Attack Decomposition

We can derive a test from the grammar by applying recursively its production rules. This procedure can be represented as a derivation tree. A derivation tree (also called parse tree) of a test is a graphical representation of the derivation steps that are involved in producing the test. In a derivation tree, an intermediate node represents a non-terminal symbol, a leaf node represents a terminal symbol, and a edge represents the applied production. Figure 6.1 depicts the derivation tree of the boolean attack: *’␣OR“a”=“a”#*. In the course of generating this test, we first apply the *<start>* rule:

$$\begin{aligned}\langle start\rangle \quad ::= \quad & \langle numericContext\rangle \mid \langle sQuoteContext\rangle \\ \mid \quad & \langle dQuoteContext\rangle \ ;\end{aligned}$$

and derive *<sQuoteContext>*. We then apply the third ruleof the grammar to derive *<squote>*, *<wsp>*, *<sqliAttack>*, and *<cmt>*. This procedure is repeated until all non-terminal symbols are derived to terminal symbols. The attack string represented by a derivation tree is obtained by concatenating the leafs from left to right.

We use derivation trees to identify which substrings of a SQLi attack are likely to be responsible for the attack being blocked or passing. Specifically, an attack is divided into substrings by decomposing its derivation tree into slices. The definition of a slice is as follows:

6.1 Definition: Slice. *A slice s is a subtree T' of a derivation tree T such that T' contains a subset of leafs of T.*

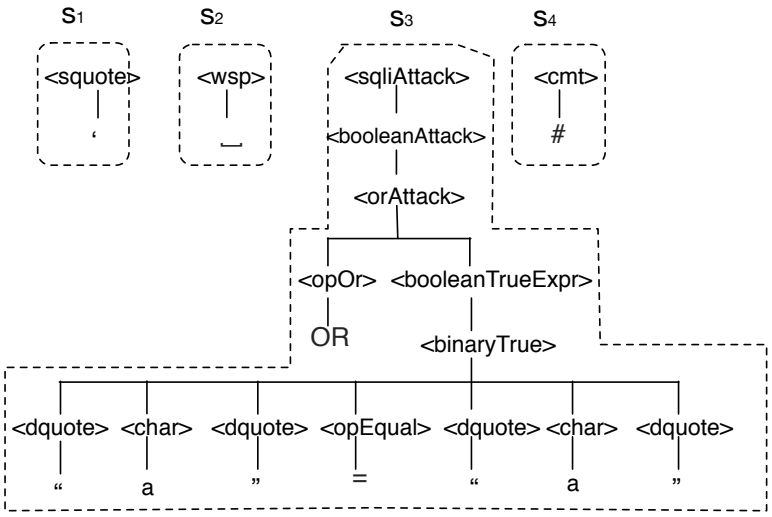


Figure 6.2. Example subset of slices decomposed from the tree in Figure 6.1.

A slice is supposed to represent a substring of an attack, hence only such subtrees of a derivation tree are considered that contain a subset of leafs. Otherwise, if the subtree contains the same leafs as the derivation tree, the represented string is not a substring, but the same string as the derivation tree. For example, for the derivation tree in Figure 6.1 the subtree with the root *<sQuoteContext>* contains the same leafs as the derivation tree and is therefore not a slice.

6.2 Definition: Minimal Slice. *A slice s is minimal if it has only two nodes: a root and only one child that is a leaf.*

The procedure to decompose a derivation tree into slices is detailed in Algorithm 2. Starting from the root node the algorithm recursively computes slices from descendant nodes by calling *VISIT(child, root, S)* in line 5. In line 11, the condition $(root.leafs \setminus node.leafs) \neq \emptyset$ ensures that only such slices are considered that comply with Def. 6.1. In line 14, the recursion ends if the node forms a minimal slice, as defined in Def. 6.2, otherwise the recursion continues.

Applying the decomposition procedure to the derivation tree in Figure 6.1 yields a set of 12 distinct slices. Figure 6.2 shows a sample of four slices decomposed from the tree.

We conjecture that the appearance of one or more slices in a test could result in the test getting blocked or bypassing. In the next sections we develop this idea further by analysing slices of a collection of tests and predicting, using machine learning, how their appearance in the tests affect their likelihood of bypassing a WAF or being blocked.

6.1.3.2 Training Set Preparation

Given a set of tests that have been labelled with their execution result against a WAF, that is a "P" or "B" label, we transform each test into an observation instance to feed our machine learning algorithm.

1. Each test is decomposed into a vector of slices $t_i = \langle s_1, s_2, \ldots, s_{N_i} RANgle$ by applying the attack decomposition procedure.

Algorithm 2 Tree decomposition into slices.

```
1:  procedure DECOMPOSETREE(root)
2:      S ← ∅
3:      children ← root.childNodes
4:      for all child ∈ children do
5:          VISIT(child, root, S)
6:      end for
7:      return S
8:  end procedure
9:  procedure VISIT(node, root, S)
10:     s ← getSlice(node)                ▷ get the slice for which node is the root
11:     if (root.leafs \ node.leafs) ≠ ∅ then
12:         S ← S ∪ s
13:     end if
14:     if s is minimal then
15:         return
16:     else
17:         children ← node.childNodes
18:         for all child ∈ children do
19:             VISIT(child, root, S)
20:         end for
21:     end if
22: end procedure
```

2. Each slice is assigned a globally unique identifier. If the same slice is part of multiple tests, the same identifier references it. We map each unique slice to an attribute (a feature) of the training data set for machine learning.
3. Every test is transformed into an observation of the training data set by checking whether the slices used as attributes are present or not in the corresponding vector of slices of the test.

As a concrete example, we have three tests t_1, t_2, t_3; the first two are blocked while the last can bypass a WAF. Their decompositions into slices and labels are shown on the left side of Table 6.1, and their encoded presentation on the right side of the table. In total, we have five unique slices from all the tests and they become attributes of the training data set for machine learning. If a slice appears in a test, its corresponding attribute value in the training data is "1", and otherwise "0".

Table 6.1. An example of test decompositions and their encoding.

t.id	vector	label
1	$\langle s_1, s_2, s_3 \rangle$	B
2	$\langle s_1, s_2, s_4 \rangle$	B
3	$\langle s_4, s_5 \rangle$	P

t.id	s_1	s_2	s_3	s_4	s_5	clz
1	1	1	1	0	0	B
2	1	1	0	1	0	B
3	0	0	0	1	1	P

6.1.3.3 Decision Tree and Path Condition

By decomposing tests into slices and transforming them into a labelled data set, we can now apply a supervised machine learning technique to predict which slices or combinations of slices are associated with tests bypassing a WAF or being blocked by the WAF. To be able to identify such slices, we rely on machine learning techniques that provide an interpretable output model. That is, the reason for the

classification of attacks into bypassing or blocked should be easily comprehensible. Therefore, we selected decision trees for this task.

In a decision tree, a node represents an attribute from a data set; each branch represents a possible value of an attribute; and a leaf node represents a classification for all instances that reach this node. In our context, each node represents a slice and the branches from the node can be "0" or "1", corresponding to whether the slice is absent or present. A leaf node classifies instances into blocked or bypassing and is labelled accordingly with "B" or "P". Figure 6.3 shows an example decision tree learned from the data in Table 6.1.

The paths from the root node of the decision tree to its leaf nodes embody the combinations of slices which are likely to be the reason for tests to bypass or to be blocked. More generally, we define a concept of *path condition* as:

6.3 Definition: Path Condition. *A path condition represents a set of slices that the machine learning technique deems to be relevant for the attack's classification into blocked or bypassing. The path condition is represented as a conjunction* $\bigwedge_i^k (s_i = val)$*, in which* $val = 1 \mid 0$*, and k is the number of relevant slices.*

The procedure for computing path conditions depends on the machine learning algorithm that is used to build decision trees. We have selected two alternative algorithms and assessed their overall impact on test results.

RandomTree. The most prominent difference to similar algorithms is that RandomTree relies on randomization for building the decision tree [Breiman, 2001]. When selecting an attribute for a tree node, the algorithm chooses the best attribute amongst a randomly selected subset of attributes. By choosing only subset of attributes, the algorithm scales well with the size of the training data set. In our context, a scalable learner is important since the data sets contain a larger number of attributes and the decision tree is frequently rebuild, as described in Section 6.1.3.4.

Given a decision tree and a slice vector V of a test t, we can obtain the path condition for t by visiting the decision tree from the root and check the presence (value = 1) or absence (value = 0) of the attributes encountered with respect to V. The procedure stops once a leaf is reached. For instance, Figure 6.3 presents a decision tree learned from the example data discussed in Table 6.1. For test t_1, the attribute s_3 is present in the test's slice vector $\langle s_1, s_2, s_3 RANgle$, thus t_1 follows the left branch and the path condition is $(s_3 = 1)$. Similarly, for test t_3 with the slice vector $\langle s_4, s_5 RANgle$, the attribute s_3 is not present, thus the right branch is followed leading to attribute s_5, which is present in the slice vector. Therefore, the resulting path condition for t_3 is $(s_3 = 0 \wedge s_5 = 1)$.

RandomForest. Machine classification methods are well-known to be unstable, that is, a small change in the training data can result in a completely different classification model [Witten and Frank, 2011]. Ensemble methods have been proposed to address the issue. In essence, multiple models are learned so that their collective predictions can mitigate bias in individual models. In this work, we implement an ensemble of classifiers to guide the test generation, i.e., instead of using only one RandomTree, we extend our technique to make use of ensembles of trees produced by RandomForest [Breiman, 2001]. However, the benefits of RandomForest come at the cost of an increased compu-

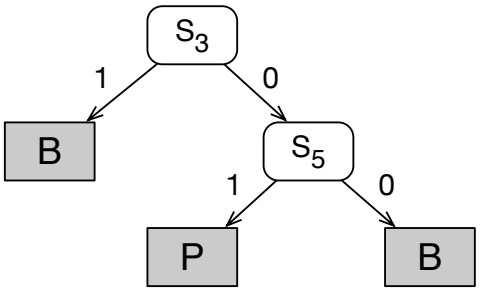


Figure 6.3. An example of a decision tree obtained from the training data in Table 6.1.

tational overhead, e.g. learning an ensemble of RandomTrees takes longer than learning only a single RandomTree. We evaluate in Section 6.2.5 whether the benefits justify the increased computational overhead.

A RandomForest consists of multiple RandomTrees. To classify an attack with the RandomForest, first each individual RandomTree classifies the attack and computes the prediction confidence (a measure for how likely the classification is correct, expressed as a probability). Then, all individual classifications are consolidated into a common consensus by computing the average of the prediction confidences for each class. Eventually, the class with the highest prediction confidence is chosen by the RandomForest as final classification.

To compute the path condition for a given attack with RandomForest, for all trees that classify the attack as bypassing a path condition is computed separately. Thereby, the path condition for each tree is computed according to the previously described procedure. The overall path condition for the entire RandomForest is, then, the conjunction of the path conditions computed from the trees.

6.1.3.4 ML-Driven Generation Strategy

Algorithm 3 details ML-Driven , our proposed machine learning based test generation strategy. From an initial test set *initTests*, which can be generated by the random attack generator from Section 6.1.2 or obtained from previous runs of this strategy, we first execute the tests against a target WAF, $execute(initTests)$, line 2. In fact, we have to execute only the tests for which the result against the WAF is not yet known. Then, the tests are transformed into the training set, $trainData \leftarrow transform(initTests)$, line 3. Details of this step are discussed in Section 6.1.3.2. A classifier *DT* is then learned from the data using either the *RandomTree* or *RandomForest* algorithm.

At line 6, the algorithm starts iterating through its main loop while the condition *not-done* holds. The steps within the loop are:

1. Rank all tests from the current test set according to their probability to bypass the WAF, $rankTests(currentTests, DT)$, line 7. The learned classifier *DT* is used to calculate the bypassing probability.
2. Until the classifier is updated, new mutants are generated as follows:
   - select the test with the highest ranking as candidate test for mutation, $t \leftarrow selectTest(currentTests)$, line 9. If more than one candidate has been ranked equally, the selection is random among them. If a test has been selected before, it will not be selected again.

Algorithm 3 ML-Driven SQLi attack generation.

```
 1: procedure MLDRIVENGEN(initTests, outputTests)
 2:     execute(initTests)
 3:     trainData ← transform(initTests)
 4:     DT ← learnClassifier(trainData)                    ▷ learn the initial classifier
 5:     currTests ← initTests
 6:     while not-done do
 7:         rankTests(currTests, DT)
 8:         repeat
 9:             t ← selectTest(currTests)
10:             V ← getSliceVector(t)
11:             pathCondition ← getPath(V, DT)
12:             s ← pickASliceFrom(V)
13:             while s ≠ null do
14:                 if satisfy(s, pathCondition) then
15:                     newTests ← mutate(t, s, MAX_M)
16:                     currTests ← currTests ∪ newTests
17:                 end if
18:                 s ← pickASliceFrom(V)
19:             end while
20:         until shouldUpdClassifier(currTests)
21:         execute(currTests)                             ▷ new tests only
22:         trainData ← transform(currTests)
23:         DT ← learnClassifier(trainData)
24:     end while
25:     outputTests ← filterBypassingTests(currentTests)
26:     return outputTests
27: end procedure
```

- get the path condition from the classifier with the slice vector V of attack t, line 11.
- pick a slice s from V and check whether it satisfies the determined path condition, $satisfy(s, pathCondition)$. If it is the case, replace the slice with an alternative slice, which also needs to satisfy the path condition, to generate new tests $newTests \leftarrow mutate(t, s, MAX_M)$.

3. When a predetermined number of mutants is generated, $shouldUpdClassifier(currTests)$ evaluates to true and the loop is exited, line 20. In the following lines, all mutants are executed, labelled with bypassing respectively blocked and added to the training data set. Finally, the classifier is retrained with the training data set.

A slice is said to satisfy a path condition if it does not affect the truth value of the condition. That is, it either appears in the predicate and complies with it, or does not appear in the predicate. In our approach, we rank tests and select those that fall in a tree leave with a high likelihood of bypassing the WAF. Therefore, we consider the path conditions of those tests to be good indicators about test content that the WAF might ignore. As a result, in our mutation step, $newTests \leftarrow mutate(t, s, MAX_M)$, we select a slice s that does not appear in the path condition of t and replace it with equivalent alternative slices that also need to satisfy the condition. Take, for example, t_2 with its slice vector $\langle s_1, s_2, s_4 RANgle$. Since its path condition is $(s_1 = 1)$, we can select s_2 or s_4 and replace them with their alternatives.

Equivalent alternatives of a slice are determined based on the root symbol of the slice and all production rules of the grammar that start with this symbol. For example, taking slice s_2 in Figure 6.2 that starts with <wsp> and derives <blank>, we obtain only one production rule from the grammar:

$$\langle wsp \rangle \quad ::= \langle blank \rangle \mid \langle inlineCmt \rangle \ ;$$

As a result, we determine only one alternative slice that starts with <wsp> and derives <inlineCmt>.

At the mutation step in line 15, $newTests \leftarrow mutate(t, s, MAX_M)$, the parameter MAX_M is an integer value that limits the number of mutants that are generated for test t and slice s. If there are more alternative slices for t and s than indicated by MAX_M, only MAX_M randomly selected alternative slices are chosen and used, in turn, to form mutants. If there are less alternative slices than indicated by MAX_M, all slices are used to form mutants.

Assuming the total test budget is constant, MAX_M controls our approach to explore the test space either *broadly* or *deeply*. When MAX_M is small, the approach generates fewer mutants per selected test, but selects more tests for mutation, thus exploring the test space in a broader fashion. When MAX_M is large, to the opposite, the approach generates more mutants per selected test, but selects fewer tests for mutation, thus exploring the test space in a deeper fashion. In the evaluation section two variants of ML-Driven are distinguished: ML-Driven B (broad, with $MAX_M = 10$) and ML-Driven D (deep, with $MAX_M = 100$). Section 6.1.4 presents a thorough analysis on how the parameter MAX_M influences the overall test results.

The classifier is regularly retrained during the course of a test run to incrementally improve its precision. At line 20, $shouldUpdClassifier$ triggers the retraining of the classifier after every 4000 generated attacks. To not exceed the resources of a typical personal computer and to finish in a reasonable time we limit the training set to 6000 blocked attacks and 6000 bypassing attacks. The mentioned values serve as defaults and can be customised by the user to match the available computing power and time constraints.

6.1.4 Enhancing ML-Driven

In this section, we analyse the differences between ML-Driven B and ML-Driven D and how they influence the efficiency of the approach. Based on the analysis, we introduce an improved variant of our attack generation strategy called ML-Driven E.

ML-Driven B and ML-Driven D differ in how the test budget is spend, or more precisely, in the value for the constant MAX_M. As explained in Section 6.1.3.4, MAX_M determines how many mutants are generated from a selected test. For ML-Driven D MAX_M is set to a lower value, which leads to ML-Driven D selecting fewer tests for mutation, but generating more mutants per selected test. For ML-Driven B MAX_M is set to a higher value, which leads to ML-Driven B selecting more tests for mutation, but generating fewer mutants per selected test. Note that for both variants the total test budget is the same, but the allocation of the test budget differs.

A detailed analysis shows that no variant is superior over the other (see Appendix A): ML-Driven D performs better at the beginning of a test run but ML-Driven B outperforms it in later stages. The

reason for this phenomenon is because of the number of bypassing tests selected and their influence on the overall performance. At the beginning, there are only a few available tests with a high bypassing probability. Due to the lower value of MAX_M, ML-Driven D selects only these tests for mutation and, thus, generates more bypassing mutants. Since MAX_M is higher for ML-Driven B, more tests are selected for mutation, resulting in selecting not only the few tests with a high bypassing probability, but also tests with a low bypassing probability. In consequence, ML-Driven B generates fewer bypassing tests. After some iterations, when more tests with a high bypassing probability are available, selecting more bypassing tests and mutating each less often proved to be more efficient in our evaluation. ML-Driven B does exactly that and yields more bypassing tests than ML-Driven D.

We address the described issue with a more flexible approach for assigning the test budget to individual tests called ML-Driven E. Instead of generating a fixed number of mutants per test, as done with ML-Driven B/D, the number of mutants is calculated dynamically. The goal of ML-Driven E is twofold: First, the available test budget should be allocated only to tests with a high bypassing probability. Second, the test budget should be divided amongst all tests that have a high bypassing probability.

Given the total mutation *budget*, a set T of tests selected for mutation, the probability $P(x)$ of test x to bypass the WAF, then the individual mutation budget m for test x is defined as:

$$m = \frac{P(x)}{\sum_{t \in T} P(t)} * budget \tag{6.1}$$

On the right-hand side, the fraction represents the relative bypassing probability of x by dividing its bypassing probability with the sum of all bypassing probabilities. By multiplying the relative bypassing probability of test x with the total budget, x is assigned a share of the total budget proportional to its relative bypassing probability.

Algorithm 4 is an extended version of Algorithm 3 and includes the proposed modification to the budget calculation.

Up to line 7, the algorithm is identical with the original algorithm (Algorithm 3). In line 8, the method $selectAttacksForMutation(rankTests)$ selects a set of attacks A that have the highest bypassing probability from all available attacks $rankTests$. All attacks above a configurable threshold, e.g. in our experiment 80%, are selected. The loop from line 9 to 21 is executed for each attack in A to generate mutants. Within the loop, in line 10 $m_t \leftarrow getMutationBudget(t, A, DF)$ calculates the individual mutation budget m_t for attack t with regards to A and DF, the latter being the classifier. This method is the implementation of Equation 6.1. Lines 11 to 20 describe the mutation procedure for t and are mostly unchanged from the original version of the algorithm, except that the number of generated mutants for t is set to m_t.

6.2 Evaluation

This section evaluates the proposed testing strategies in two separate case studies: a popular open-source WAF and a proprietary WAF that protects a financial institution. Section 6.2.1 introduces the

Algorithm 4 ML-Driven E.

```
1:  procedure MLDRIVENGEN(initTests, outputTests)
2:      execute(initTests)
3:      trainData ← transform(initTests)
4:      DF ← learnClassifier(trainData)                    ▷ learn the initial classifier
5:      currTests ← initTests
6:      while not-done do
7:          rankTests(currTests, DF)
8:          A ← selectAttacksForMutation(rankTests)
9:          for all t ∈ A do
10:             m_t ← getMutationBudget(t, A, DF)
11:             V ← getSliceVector(t)
12:             pathCondition ← getPath(V, DF)
13:             while m_t > 0 do
14:                 s ← pickASliceFrom(V)
15:                 if satisfy(s, pathCondition) then
16:                     newTests ← mutate(t, s)
17:                     currTests ← currTests ∪ newTests
18:                     m_t = m_t − 1
19:                 end if
20:             end while
21:         end for
22:         execute(currTests)                             ▷ new tests only
23:         trainData ← transform(currTests)
24:         DF ← learnClassifier(trainData)
25:     end while
26:     outputTests ← filterByPassTests(currentTests)
27:     return outputTests
28: end procedure
```

case studies. Section 6.2.2 formulates the research questions. Section 6.2.3 explains the procedure we followed to execute the experiments and Section 6.2.4 the measured variables. Finally, in Section 5.3.4 the results are presented.

6.2.1 Subject Applications

6.2.1.1 Open-Source WAF

In this case study, the firewall under test is *ModSecurity*, which implements the OWASP core rule set. ModSecurity is an open-source web application firewall that can be deployed with the Apache HTTP Server to protect web applications hosted under the server. Depending on the applications under protection, different firewall rule sets defined for different purposes can be used. The OWASP core rules target various kinds of attacks, e.g. Trojan, Denial of Service, and SQL Injection, and is maintained by an active community of security experts.

The web applications under protection are HRS, Cyclos, and SugarCRM. HRS is a service-oriented based system, providing web services for room reservation. It was developed and used

in [Coffey et al., 2010a]. Cyclos is a popular open-source Java/Servlet Web Application for e-commerce and online payment[3]. SugarCRM is a popular customer relationship management system[4]. SugarCRM and Cyclos have been widely used in practice. In our experiment setting, the three applications are deployed on an Apache HTTP Server under Linux. ModSecurity is embedded within the web server; it protects the application's web services from SQLi attacks. Specifically, since these web services receive SOAP messages[5] from web clients, a malicious client can seed a SQLi attack string into a SOAP message and submit it to the web services in order to gain illegal access to data or functionality of the system.

In this chapter, note that our testing target is the WAF that protects the applications, not the applications themselves, as our focus is on testing firewalls. HRS, SugarCRM, and Cyclos play solely the role of a destination for SQLi tests that bypass the WAF.

6.2.1.2 Proprietary WAF

In the industrial setting, we evaluate our approach on a proprietary WAF that is used in a corporate IT environment to protect back-end web services. To provide protection from malicious requests, the WAF validates incoming requests in two steps: First, the values in a request are validated with respect to data types (e.g. string or numeric) and boundary constraints, e.g. a credit card number is expected to be a sequence of 16 to 19 digits. In a second step, each value is checked to make sure that it does not contain known malicious string patterns (i.e., using a SQLi blacklist) commonly used in attacks. Only if the request passes both validation steps the request is forwarded to the back-end services.

The web services under protection are the backbone of a financial corporation. They process thousands of transactions daily. Clients interact with the web services using SOAP.

To evaluate our approach with the proprietary WAF, because of the massive number of test executions required by our experiments, we had to make use of a high-performance cluster [Varrette et al., 2014] and, furthermore, we had to optimise a replica of the test environment to significantly decrease response times when invoking services.

In our optimised test environment, all configurations related to request filtering, that is whether a request is blocked or let through, are copied. Other configurations, e.g. logging or encryption, are disabled. The web application under protection is replaced by a simple mock-up application, which implements the same web service interface as the original application under protection. The mock-up replays a set of recorded responses from the original application and, thus, the WAF remains unaffected. Table 6.2 shows the message round trip time (RTT) per operation computed over a time span of 30 days with the actual environment (RTT_{ACT}) compared to the optimised environment on the HPC (RTT_{HPC}). As we can see, the time has been reduced significantly. Note that, even with these optimisations, the total computation time of our experiments is equivalent to 8 years, 337 days, and 12 hours on a single CPU core.

Even though an optimised test environment is required from an experimental standpoint, in practice, testing the firewall is just a single test activity in an array of test activities (e.g., testing the WAF,

[3] http://project.cyclos.org

[4] http://sourceforge.net

[5] http://www.w3.org/TR/soap12-part1

services, front end). Given time and resource constraints, creating and maintaining test environments that are specific for each single test activity is very costly. Based on our experience, we found that test engineers prefer to test copies of the actual WAF configuration and services.

Since we, by design, optimised our test environment to enable large scale experiments, the test execution times in our experimental setting is not representative of test execution times in the actual environment (see Table 6.2). This is a problem as it biases the results of our experiments to the advantage of approaches that are less expensive in terms of test case generation but lead to the execution of more test cases, such as random testing. Therefore, in our analyses, we transform the time scale of the optimised environment into a realistic one, accounting for the actual test execution times in practice.

Assume $T = \{t_1, \ldots, t_n\}$ is a set of timestamps measured on the experimental environment, such that one timestamp is noted after the execution of every test. Then, for the i-th test case execution, $f(t_i) = t_i + (RTT_{ACT} - RTT_{HPC}) * i$ transforms a timestamp $t_i \in T$ into the corresponding timestamp in the actual environment ($f(t_i)$). We will use the latter time scale to compare test strategies in a realistic fashion.

Table 6.2. Average response time in milliseconds of some web service operations in our experimental environment compared to the case study's environment.

	Op. 1	Op. 2	Op. 3	Op. 4
RTT_{HPC}	11,36	11,39	11,46	16,61
RTT_{ACT}	456,56	180,02	302,09	854,63

6.2.2 Research Questions

This work investigates several variants of a machine learning-driven testing strategy and a random testing strategy. We compare and evaluate all these strategies for their capability of finding bypassing attacks on both subject applications, i.e. ModSecurity and a proprietary WAF.

Since all testing strategies generate attacks from the same input space, i.e. the grammar introduced in section 6.1.1, we evaluate how efficient the different strategies are in sampling the input space for bypassing attacks. Therefore, we measure for each strategy how many distinct bypassing attacks are found over time.

RQ1: *How efficient are ML-Driven E, ML-Driven B, ML-Driven D, and RAN in finding bypassing tests?*

To assess the impact of machine learning on the test result, we implemented two alternative classifiers for each of the ML-Driven strategies: *RandomTree* and *RandomForest* (see Section 6.1.3.3). The RandomTree algorithm is selected because it can handle large data sets with many instances and attributes faster than comparable algorithms, i.e., C4.5 [Quinlan, 1993]. In our context, this is an important property because, during the course of a test run, the ML-Driven techniques frequently rebuild the classifier to include new bypassing tests in the training set.

The RandomForest algorithm is also considered because it computes an ensemble of classifiers and, thus, is expected to be more robust against changes in the training set. Ensemble methods avoid

the instability known to some machine learning methods, that is, a small change in the training data resulting in a significantly different classification model. Ensemble methods, like the RandomForest, tackle this problem by learning multiple models so that their collective prediction can avoid biases in individual models. However, *RandomForest* comes at a higher computational cost than *RandomTree*, since multiple models have to be learned.

RQ2: *Does the choice of machine learning algorithm matter?*

We evaluate whether, in our context, the benefits of the RandomForest compared to the RandomTree justify the increased computation overhead to learn the classifier. Therefore, we compare how many bypassing tests are found over time with these two algorithms. In addition, we assess whether the algorithms have an impact on the stability of the test result, i.e. we compare the variation among repetitions of the same test run.

RQ3: *Does the machine learning classifier's accuracy improve over retraining iterations?*

The classifier is regularly retrained during the course of a test run with the goal to incrementally improve its accuracy. RQ3 assess the gained accuracy by comparing the F-measure of classifiers obtained in consecutive training iterations.

RQ4: *Are we learning new, useful attack patterns as the number of bypassing attacks increases?*

RQ4 assesses whether, as we find more bypassing tests, we also identify more attack patterns that can be useful to improve the rule set of the WAF. In our context, an attack pattern is the underlying root cause that enables an attack to bypass the WAF. For example, the attacks *union *!50000*select pwd from user* and *union *!50000*select 99* share the pattern *union *!50000*select* (root cause) while the remainder of the attacks differ. We want to investigate whether identifying successful attack patterns helps understanding why attacks are bypassing and eventually fix the WAF to correctly detect further attacks.

In the ML-Driven techniques, such a pattern is characterised by a path condition (see Def. 6.3). A path condition characterises the slices, or combination of slices, that are likely causing an attack to bypass. Thus, a path condition represents a pattern that is not correctly detected by the WAF.

To answer RQ4, we measure how many path conditions can be extracted from a model that is learned by the ML-Driven techniques. More specifically, we analyse the growth in the number of path conditions as the number of successful, distinct attacks grows over time.

6.2.3 Procedure

We implemented the techniques proposed in this work into our SQLi testing tool called *Xavier*. *Xavier* supports the automated testing of web services for SQLi vulnerabilities and has been de-

scribed in [Appelt et al., 2014]. ML-Driven generates test cases in the form of SQLi attack strings, such as *'␣OR"1"="1"#*. To generate malicious requests, *Xavier* takes such attack strings and injects them into sample SOAP messages, which are subsequently sent to an application under test. *Xavier* therefore relies on sample SOAP messages as inputs. They can be taken from existing web service test suites, or can easily be generated from the WSDL[6] that describes the service interface under test.

In our experiments, when available, we use SOAP messages from the functional test suite of the service under test (Cyclos and our industrial case study) or, otherwise, we manually create SOAP messages from the WSDLs (HRS, SugarCRM). Each of these messages consists of a number of parameters and their legitimate values, which the services expect and that the WAF has to let through. In our testing process, each SOAP message is considered separately. A test generation technique, ML-Driven or RAN, continuously generates attacks, injecting one attack each time into a parameter of the selected SOAP message to create a new SOAP message, and then sends it to the web server.

Incoming SOAP requests to the web server are first treated by the WAF and only those that comply with firewall rules are forwarded to web applications, and otherwise are blocked. In case a request is blocked, the WAF replies to the client that issued the request with a special response, stating that the request has been denied. When our testing tool, *Xavier*, receives such a response, it marks the test, embedded in the original request, with a blocked label "B" (for blocked), and otherwise a passed label "P" (bypassing).

6.2.4 Variables

The following variables are controlled or measured in our experiments:

D_t: The number of distinct tests that can bypass a target WAF at time t is a way to measure the efficiency of a test strategy. Note that, given two distinct tests in the same attack category, one might be caught by the WAF while the other bypasses it. This may be caused by tests in a category that should be handled by different rules, some of them missing or incorrect in the current WAF rule set, or by an identical rule that is not general enough to block all tests in a category. Therefore, identifying similar but distinct bypassing tests is useful to identify attack patterns.

D_{pc}: The number of distinct path conditions that can be extracted from decision trees. Each path condition characterises a string pattern that a group of bypassing attacks has in common. Such string patterns can be added to the WAF rules to prevent further attack attempts containing the same pattern. Hence, D_{pc} is a measure for how many attack patterns have been uncovered.

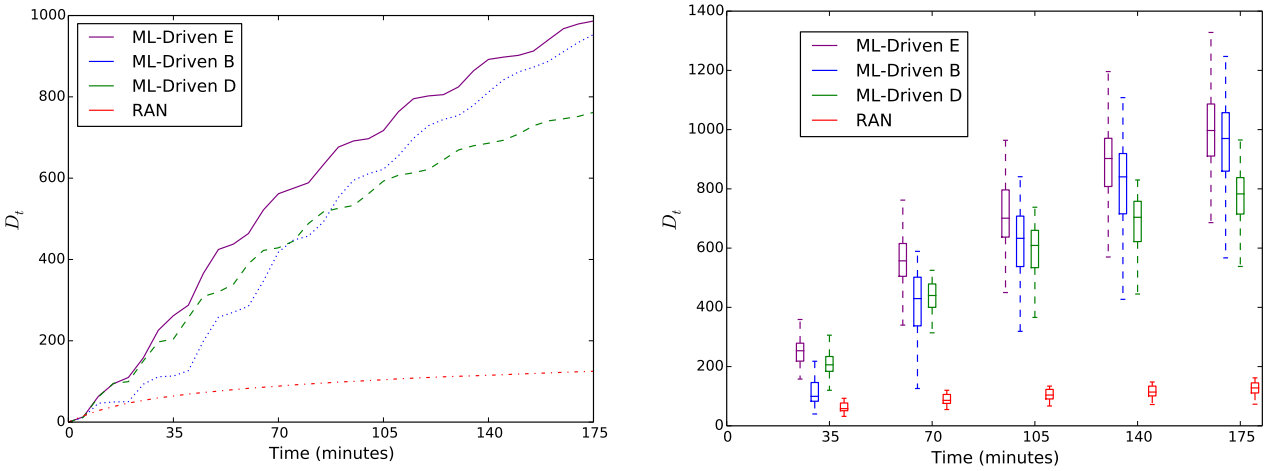
6.2.5 Results

6.2.5.1 Performance Comparisons

To answer *RQ1*, we applied the testing techniques to both subject applications and measured how many bypassing tests were found over time (D_t). To account for the randomisation involved in the testing techniques, we repeated each test run 10 times. Each time a new test was generated, we noted the passing wall-clock time since the beginning and then executed it to see whether or not it could

[6] http://www.w3.org/TR/wsdl

bypass the WAF. We compared the performance of the techniques based on the cumulative number of distinct bypassing tests generated over time.



(a) Average number of bypassing tests (D_t)

(b) Statistical variation for D_t

Figure 6.4. Number of bypassing tests (D_t) found over time for all tested parameters (10 repetitions each) for ModSecurity.

For ModSecurity, we selected randomly nine parameters in total for testing, three parameters from each web application (HRS, SugarCRM, and Cyclos). Figure 6.4a depicts the average number of distinct, bypassing tests generated over time for ModSecurity. This is an average over 10 repetitions on nine SOAP parameters for each technique, measured within intervals of five minutes (the test results for each individual parameter are in Appendix B.1). Figure 6.4b depicts the same data as boxplots to help visualise statistical variation.

The first observation is that all techniques can generate tests that bypass the WAF, suggesting that the WAF does not provide complete protection from SQLi attacks, putting online systems under its protection at risk. Further, by observing executed SQL statements on the database, we found that these bypassing tests can exploit SQLi vulnerabilities in HRS and SugarCRM. Second, the sharply increasing plots corresponding to ML-Driven E, ML-Driven B and ML-Driven D indicate that they are much more efficient than RAN, the baseline for comparison. Overall, the results show that the ML-Driven techniques outperform RAN by an order of magnitude with respect to the number of distinct bypassing tests generated.

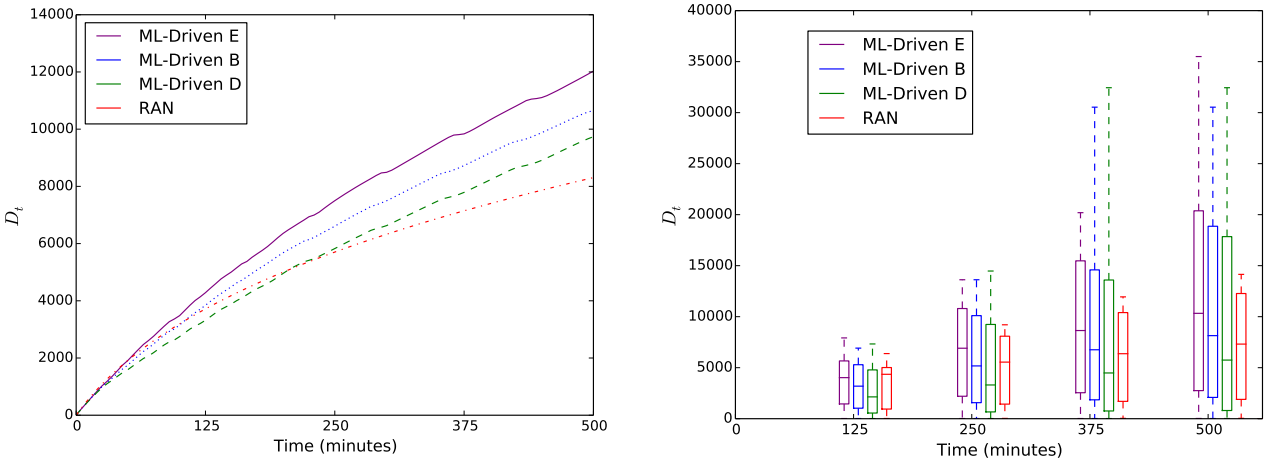
Among the machine learning-driven techniques, ML-Driven E constantly finds the most bypassing tests compared to ML-Driven B and ML-Driven D, which suggests that the concept of a more flexible budget allocation works well. One issue with ML-Driven D/B is that at the beginning of the test run, ML-Driven D finds more bypassing tests, while this is the opposite later in the test run. Since both techniques implement the same algorithm, this phenomenon can be attributed to a difference in the choice of parameters, or, more precisely, the parameter that determines the number of mutants generated per test (a detailed analysis is provided in a Appendix A) While ML-Driven D and B generate a fixed number of mutants per test, ML-Driven E adjusts the number of mutants in proportion to the test's bypassing probability. As a result, ML-Driven E spends the test budget more efficiently and finds bypassing attacks faster.

The plots for the ML-Driven techniques are also slightly oscillating, thus depicting the effect of iterative re-training of the classifier. The flat segments match the time intervals where the classifier is recomputed and no new tests are generated. The slopes of the plots tend to decrease over time as it becomes increasingly harder to find new bypassing tests that have not yet been executed.

We now address *RQ1* for the second subject application, the proprietary WAF. As for ModSecurity, all techniques are evaluated using how many bypassing tests are found over time.

Due to the considerably higher complexity of the WAF in the industrial case study, we selected a higher number of parameters for testing. Given that all testing strategies have to be applied to each parameter and each test run is repeated 10 times, testing all parameters is infeasible. Therefore, we selected one parameter for each distinct data type in the WSDL of the services under test, which results in a total of 75 parameters. The WAF in this case study determines the input validation routine to be executed for a parameter based mainly on the corresponding data type; hence selecting one parameter per data type maximises the coverage of input validation routines.

Out of the 75 tested parameters, bypassing tests could be generated for 29 parameters. Each testing technique is able to generate bypassing tests for all of these 29 parameters. Figure 6.5a depicts the average number of distinct bypassing tests per test strategy. The average is computed from all tested parameters and 10 repetitions per parameter.



(a) Average number of bypassing tests (D_t)

(b) Statistical variation for D_t

Figure 6.5. Number of bypassing tests (D_t) found over time for all tested parameters (10 repetitions each) for the proprietary WAF.

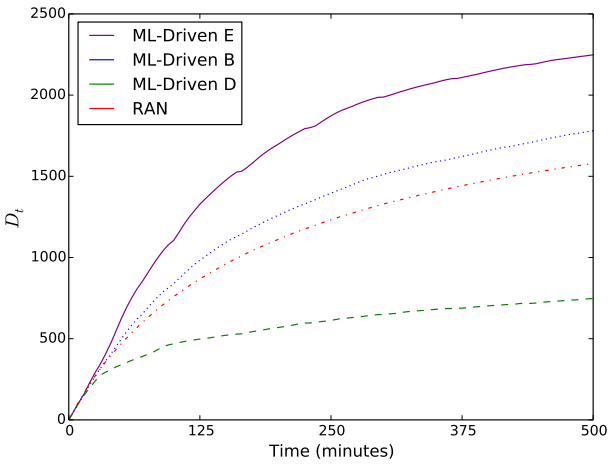
Out of all techniques, ML-Driven E generates the most bypassing tests on average, followed by ML-Driven B and ML-Driven D. RAN finds the least bypassing tests. Since the statistical variation in the plots is high (see boxplot in Figure 6.5b), we separate the parameters into groups to better analyse the results. Parameters in the same group share similar input constraints in terms of number of allowed characters and tend to produce a similar number of bypassing test cases. It is worth noticing that this particular WAF considers not only such input constraints but also other criteria (e.g., SQLi blacklist). Therefore, parameters in a same group do not necessarily produce the same results. Table 6.3 shows the groups: Group 1 has two parameters that share an input constraint that restricts the number of characters to a maximum of eight. Similarly, Groups 2, 3, and 4 have similar constraints with 16, 25,

and 35 characters, respectively.

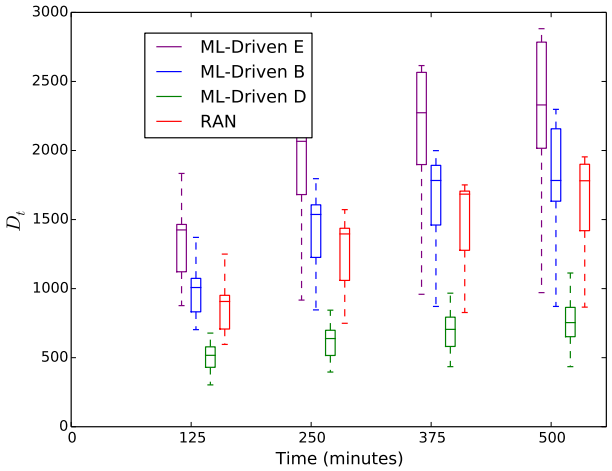
Table 6.3. Groups of parameters with a similar input constraint.

	#Parameter	Input Constraint
Group 1	2	Up to 8 Char.
Group 2	7	Up to 16 Char.
Group 3	8	Up to 25 Char.
Group 4	12	Up to 35 Char.

From the plots for Groups 2, 3, and 4, an interesting trend can be observed. The more bypassing tests for a parameter, the smaller the difference in test results across ML-Driven techniques. For example, ML-Driven E is the most efficient in Group 2; ML-Driven B is the second most efficient and ML-Driven D is by far the least efficient. For Group 3, the gap between ML-Driven D and ML-Driven E/B decreases over time. Finally for Group 4, there is no significant difference across ML-Driven techniques.

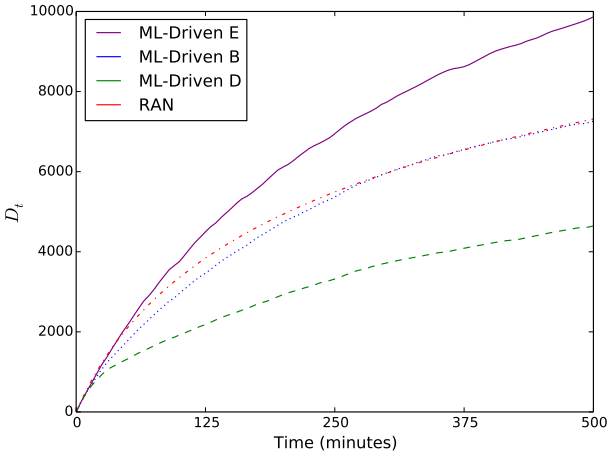


(a) Average number of bypassing tests (D_t)

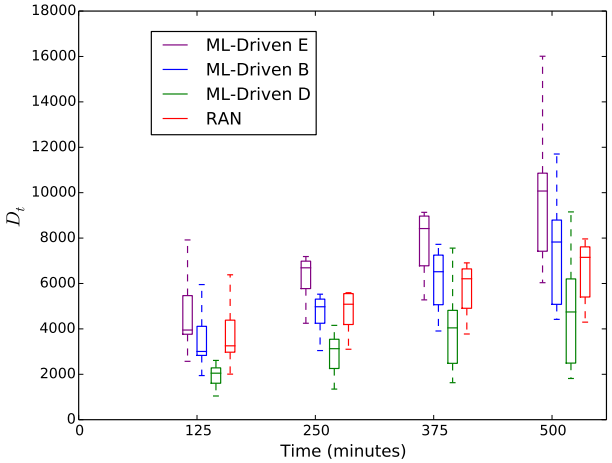


(b) Statistical variation for D_t

Figure 6.6. Group 2: Number of bypassing attacks (D_t) found over time, proprietary WAF, 7 parameters.

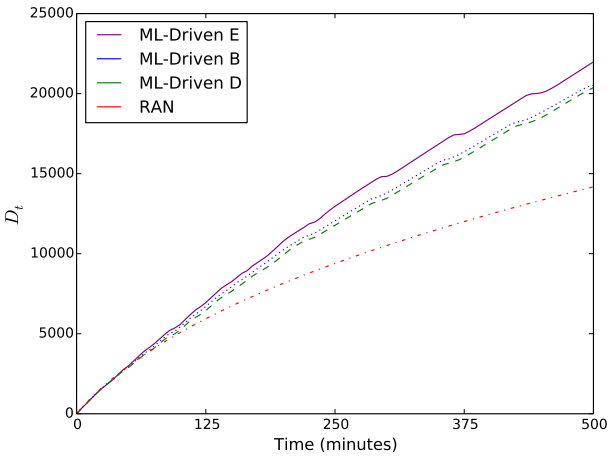


(a) Average number of bypassing tests (D_t)

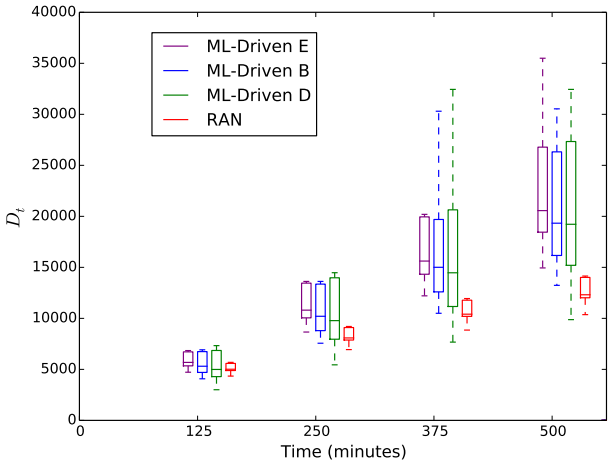


(b) Statistical variation for D_t

Figure 6.7. Group 3: Number of bypassing attacks (D_t) found over time, proprietary WAF, 8 parameters.



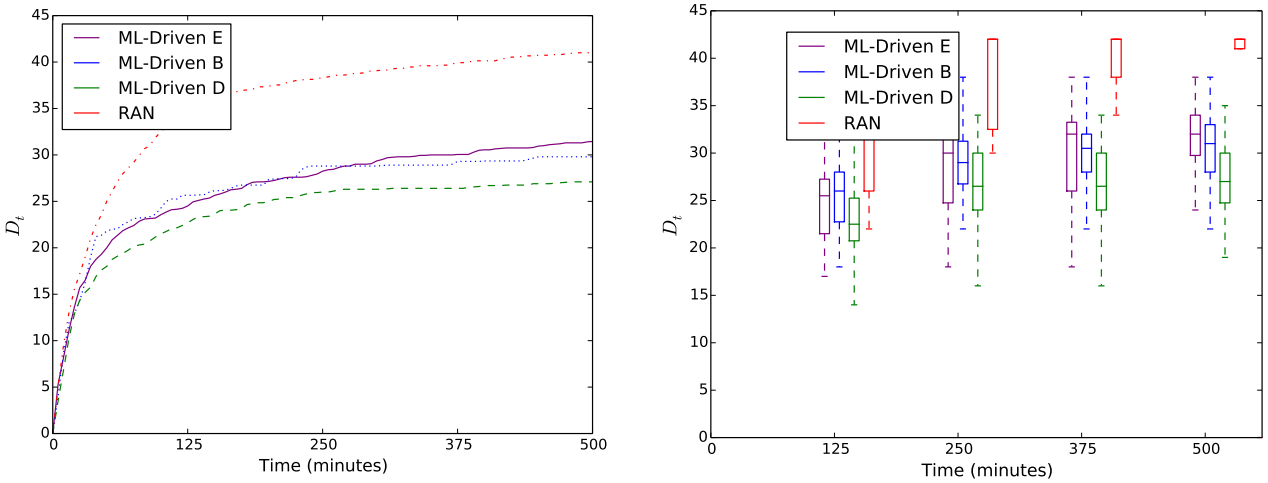
(a) Average number of bypassing tests (D_t)



(b) Statistical variation for D_t

Figure 6.8. Group 4: Number of bypassing attacks (D_t) found over time, proprietary WAF, 12 parameters.

The average number of bypassing tests for the two parameters in Group 1 is depicted in Figure 6.9a. All techniques tend to saturate after about 125 minutes and, after that, the number of bypassing tests increases very slightly. RAN is the most efficient, reaching up to approximately 40 bypassing tests. All the ML-Driven techniques perform in a similar fashion, finding around 30 bypassing tests. The most prominent difference of this group, as compared to the others, is that very few bypassing tests are found. This is due to the fact that the number of allowed characters is only eight, thus resulting in a small number of attacks to bypass. The test results for Group 1 show that RAN performs better than ML-Driven techniques.



(a) Average number of bypassing tests (D_t)

(b) Statistical variation for D_t

Figure 6.9. Group 1: Number of bypassing attacks (D_t) found over time, proprietary WAF, 2 parameters.

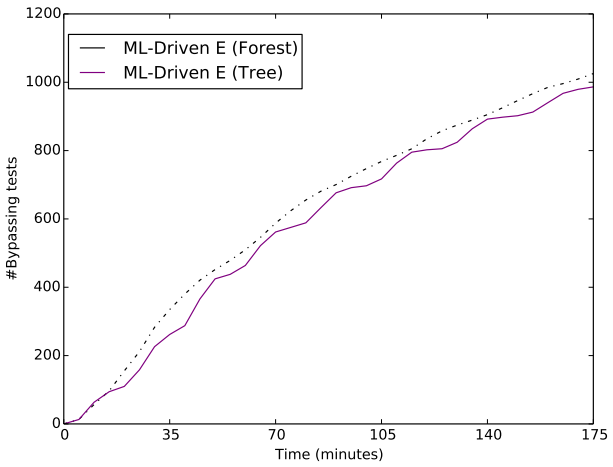
6.2.5.2 Influence of the machine learning algorithm on the test results

RQ2 assesses how the chosen machine learning algorithms, i.e. RandomTree or RandomForest, influence the test results. To answer this research question we run ML-Driven E, which proved to be the most efficient variant of ML-Driven, with both RandomTree and RandomForest, and compare the results. The evaluation is performed on the open-source case study (ModSecurity) in the same fashion as for RQ1.

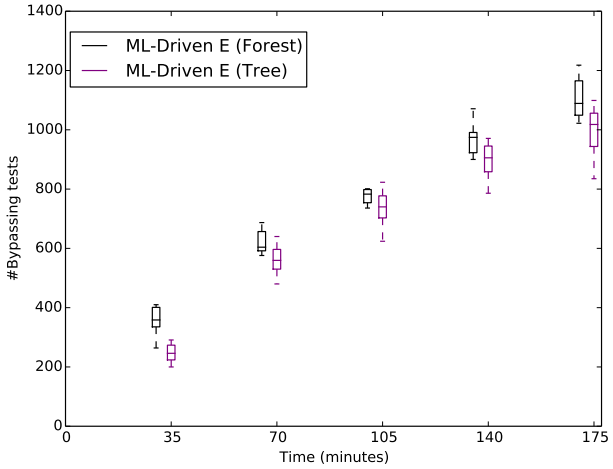
Figure 6.10a shows the average number of bypassing tests for nine selected parameters and 10 repetitions. The result indicates that ML-Driven E with RandomForest finds slightly more bypassing tests than ML-Driven E with RandomTree. However, the difference is not practically significant. The advantage of using a more stable classifier is partially lost due to the increased computation time for constructing this classifier.

Figure 6.10b shows a sample boxplot corresponding to the observed statistical variation for one representative parameter over 10 repetitions. We can see that the variation is small and similar for RandomForest and RandomTree. Similar results are observed for the other parameters. Therefore, to conclude, the choice of machine learning algorithm has no significant impact with respect to the number of bypassing tests or the degree of variation among repetitions.

RQ4 considers both RandomForest and RandomTree, in analysing the number of obtained path



(a) Average number of bypassing tests found for nine tested parameters (10 repetitions each) in ModSecurity.



(b) Boxplots for the number of bypassing tests found for one representative parameter (get-relationships) in ModSecurity.

Figure 6.10. Number of bypassing tests found with ML-Driven E using the RandomTree algorithm compared to ML-Driven E using the RandomForest algorithm.

conditions.

6.2.5.3 Assessing the impact of iterative retraining on the classifier's accuracy

The classifier is regularly retrained during the course of a test run with the goal to incrementally improve its accuracy. In this section, the classifier's accuracy over iterations is assessed by comparing the F-measure of classifiers obtained in consecutive training iterations.

The parameter K of the *RandomTree* algorithm has a strong influence on the accuracy of the learned classifier. K determines the percentage of the attributes that the algorithm considers when building each decision tree node. When K is larger, more attributes are considered and the learned classifier is likely to have a higher accuracy. However, a larger K also comes at a higher computational cost for building the decision tree. Therefore, the choice of parameter K is a trade-off between the classifier's accuracy and its computational cost. To find an optimal choice for K in the context of the presented problem, the *RandomTree* algorithm is run with different values for K and the F-measure of the learned classifiers is compared.

Besides F-measure, M_{size} is also an important quality of a classifier since it affects the cost of using the classifier. When M_{size} is larger, more computation time is required for ranking tests and checking path conditions. Therefore, ideally, we prefer classifiers that are small (low M_{size}), accurate (high F-measure), and that are less expensive to compute (low K).

Figure 6.11 depicts the average F-measure for class "P" and the average M_{size} over iterations. Only plots for K equals to 20%, 40%, and 60% are displayed to avoid cluttering the figures. Plots for K equals to 10%, 30%, and 50% share similar trends and, thus, do not affect the interpretation of the results.

The results confirm the expectation that a higher K yields a higher F-measure and a lower M_{size}. With an increase in K the F-measure and M_{size} are converging. For example, the difference between

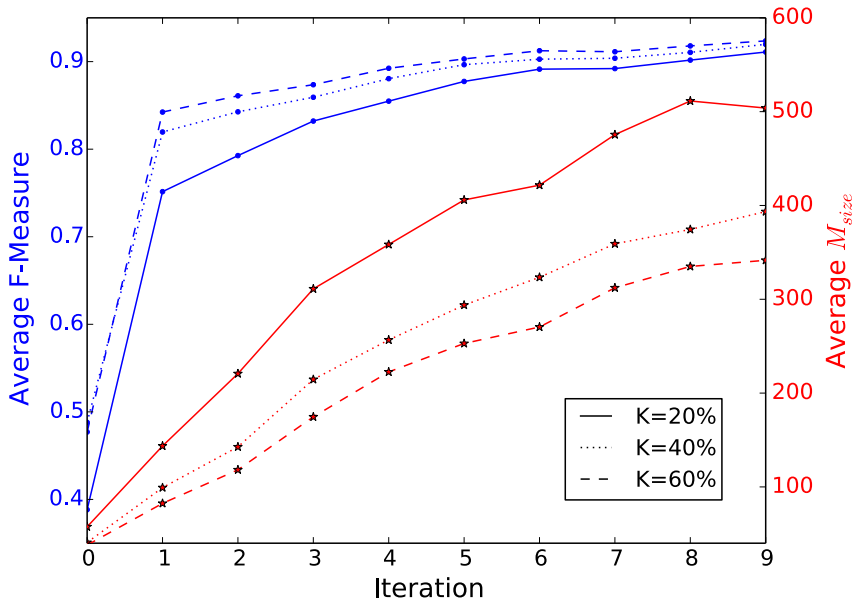


Figure 6.11. Average F-measure of class "P" (left Y-axis) and average model size (M_{size}, right Y-axis) for different K values over iterations. The data were obtained from 20 repetitions.

K=20% compared to K=40% is considerably larger than between K=40% and K=60%. In addition, though not shown on the figures, F-measure for class "B", which is the majority class in training data, remains constantly high (above 99%) regardless of K. Moreover, the results suggest that the accuracy of the classifier does significantly improve over iterations, especially in the first four or five iterations, though their size also grows steadily until iteration 10.

To conclude, K values within the range 40%±10 should be used for *RandomTree* in our context and the training process should go, as a ballpark figure, through a minimum of 5 to 10 iterations. Setting K around 40% produces classifiers that are a good trade-off between F-measure, M_{size}, and the computational cost to build the classifier. Increasing K above that range would require more time for training the classifier while bringing very little accuracy improvement. Note that all other experiments in this chapter report only results for K equal to 40%.

6.2.5.4 Learning useful attack patterns

This section investigates how, and to which extent, ML-Driven techniques help identify attack patterns, which are responsible for the bypassing attacks. As explained in Section 6.1.3, the ML-Driven techniques compute a model (i.e decision tree(s)) that is used to guide the generation of new attacks. However, this model can also be used to abstract common string patterns shared by and possibly causing bypassing attacks.

To answer RQ4, we analyse the models that are learned during the test runs of ML-Driven E. We use and compare the path conditions from test runs with both alternative machine learning algorithms, RandomForest and RandomTree.

Figure 6.12 depicts the average number of path conditions that can be extracted from a model (red) and the number of bypassing tests with which the model is trained (blue). Since the ML-Driven techniques retrain the model in fixed intervals, the corresponding training iterations are depicted on

the horizontal axis. These curves are based on the average of all test runs performed with the ModSecurity case study.

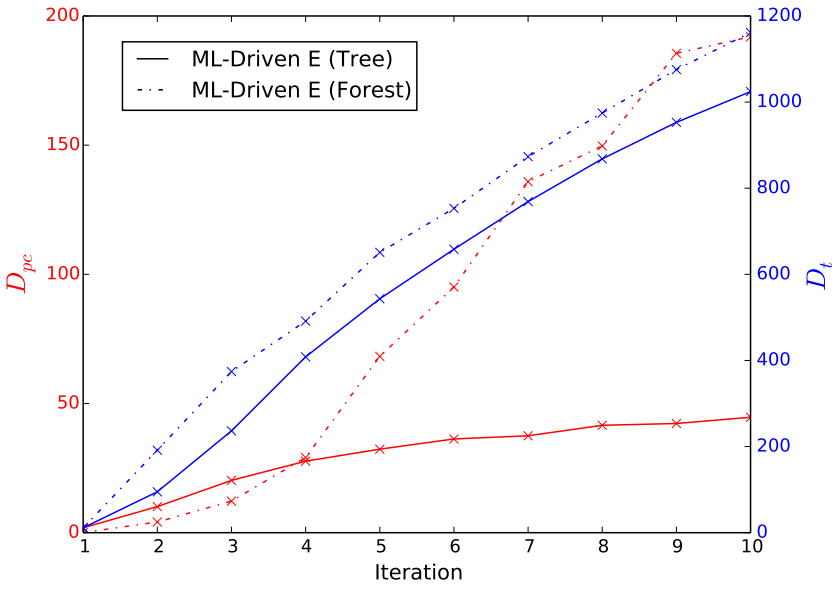


Figure 6.12. Number of path conditions (red y-axis on the left) and bypassing tests (blue y-axis on the right) for ML-Driven E with RandomTree and RandomForest.

The number of distinct, bypassing tests, which are used to train the model, is steadily increasing for both techniques. This is due to fact that the further a test run progresses, the more bypassing tests are found and are used, in turn, to retrain the model. The number of path conditions that can be extracted from a model is also steadily increasing for both techniques, although beginning from iteration 4, there are significantly more path conditions extracted with RandomForest than with RandomTree. For RandomForest, the number of path conditions is close to 200 after 10 iterations, compared with 50 for RandomTree.

For both techniques the number of bypassing tests used to train the model and the number of path conditions obtained from the model are concurrently increasing. This is no surprise as more test cases lead to refined models. For RandomForest, that leads to even more path conditions since many more trees are built.

6.2.5.5 Understanding bypassing attack patterns

A path condition can help determine the reason why a set of attacks is bypassing. This knowledge can be utilised to stop further attack attempts from succeeding by inferring a string pattern from the path condition that matches all the attacks described by it. Such a string pattern can be in turn added to the WAF's rule set to block all further attacks containing the same string pattern.

Let us consider an example of a path condition: $pc_1 = S1d \wedge Sf1 \wedge S5 \wedge \neg S11 \wedge \neg S25 \wedge \neg S26 \wedge \neg S1b \wedge \neg S15 \wedge \neg S2f$.
Table 6.4 shows the literals that are represented by the corresponding slices. An example attack characterised by pc_1 is *0/**/or 1#*, which is an obfuscated variation of a tautology attack, e.g. *" or 1=1*. All attacks characterised by pc_1 contain the slices $S1d$, $Sf1$ and $S5$. If a WAF blocks any input

containing these three slices, the attacks characterised by pc_1 do no longer bypass.

Table 6.4. Slice encodings.

Slice	Literal		Slice	Literal
$S1d$	or		Sb	!
$Sf1$	or 1		$S3$	1
$S5$	/**/		$S6$	~1
$S26$	#		$S57$	␣1
$S2$	␣		$S29$	␣0
$S0$	0		$S15$	)
Sf	true			

During test execution, the number of discovered path conditions increases. With more path conditions, the reasons why attacks are bypassing can be more precisely characterised. To illustrate this phenomenon, we analyse the path conditions obtained in different iterations of the same test run. To limit the number of path conditions in this example, we only consider path conditions that contain the slice $S1d$, like in pc_1. This slice represents the SQL keyword *or* and can for example be part of a SQL tautology attack, e.g. *" or 1=1*.

The previously analysed path condition pc_1 is obtained from the first iteration of a test run. In contrast, Table 6.5 shows more path conditions from the same test run obtained after the fifth iteration. Table 6.6 depicts the string patterns that can be devised from the path conditions.

Table 6.5. Path Conditions from iteration 5.

Id	Path condition
pc_2	$S1d \wedge S57 \wedge \neg S25 \wedge \neg S26 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1 \wedge \neg S3e \wedge \neg S0 \wedge \neg S38 \wedge \neg S29 \wedge \neg S5 \wedge \neg Se1 \wedge \neg S1c \wedge \neg S1de \wedge \neg S76 \wedge \neg S67$
pc_3	$S1d \wedge S26 \wedge S15 \wedge \neg Sf \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_4	$S1d \wedge S29 \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S38 \wedge \neg S67$
pc_5	$S1d \wedge Sf \wedge S26 \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg Sc \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_6	$S1d \wedge S0 \wedge \neg S25 \wedge \neg S26 \wedge \neg Sc \wedge \neg S14 \wedge \neg S3e \wedge \neg S38 \wedge \neg S29 \wedge \neg S5 \wedge \neg Se1 \wedge \neg S1c \wedge \neg S1de \wedge \neg S76 \wedge \neg S67$
pc_7	$S1d \wedge Sc \wedge \neg S1c \wedge \neg Se1 \wedge \neg S25 \wedge \neg S5 \wedge \neg S14 \wedge \neg S1de \wedge \neg S3e \wedge \neg S76 \wedge \neg S29 \wedge \neg S38 \wedge \neg S67$
pc_8	$S1d \wedge S5 \wedge S26 \wedge S2 \wedge S0 \wedge Sb \wedge \neg Sa8 \wedge \neg S7c3 \wedge \neg Sf \wedge \neg S1e1 \wedge \neg S25 \wedge \neg Sd \wedge \neg Sc \wedge \neg S14 \wedge \neg S15 \wedge \neg S26d \wedge \neg S60 \wedge \neg S2e \wedge \neg S2d \wedge \neg S38 \wedge \neg S3$
pc_9	$S1d \wedge S3 \wedge S5 \wedge S6 \wedge \neg Sf \wedge \neg S25 \wedge \neg Sf1 \wedge \neg Sc \wedge \neg S26 \wedge \neg S1b \wedge \neg S5a \wedge \neg S14 \wedge \neg S71 \wedge \neg S57$

The first notable difference between both sets of path conditions is their number. In the first iteration, only one path condition contains Slice $S1d$ while there are eight of them by the fifth iteration. When analysing the patterns identified by the path conditions, there are several interesting observations. First, pc_2 to pc_7 identify patterns of bypassing attacks that use Slice $S1d$, but require $S5$ to be

absent ($\neg S5$), thus suggesting the pattern identified in the first iteration was incomplete. For example, pc_5 characterises bypassing attacks containing slice $S1d$ combined with the boolean identifier *true* and the comment symbol #, e.g. *" or true#*. Such attacks are not matched by the pattern represented by pc_1 and, thus, would still bypass the WAF if only pc_1 would be considered in fixing the WAF.

Given the learned patterns in Table 6.6, a simple strategy to stop attacks containing the SQL keyword *or* might be first to identify the string that appears the most frequently in the patterns and to block any input containing that string. In this example, the string *or* appears in all patterns. However, blocking every input containing *or* would lead to many false positives. Therefore, *or* should be combined with other frequently appearing strings. The second most frequent string is the SQL comment character #. By blocking all inputs that contain both strings, *or* and #, the likelihood of false positives is reduced and all attacks characterised by pc_3, pc_5, and pc_8 are blocked. In a similar fashion, the attacks characterised by the remaining path conditions can be blocked by filtering inputs containing a combination of *or* and one of the characters *"*, */**/*, 1, and 0.

The procedure of combining slices across path conditions is repeated until an acceptable rate of false positives is achieved. Note that this strategy does not make use of the negated slices ($\neg S..$) of the path conditions. The absence of the slices was, however, part of the indicators why attacks can bypass the WAF. Therefore, they are already recognised by the existing firewall rule set.

Table 6.6. Learned patterns.

	Identifier	Pattern
Iteration 1	pc_1	*or␣1, /**/*
Iteration 5	pc_2	*or, ␣1*
Iteration 5	pc_3	*or,), #*
Iteration 5	pc_4	*or, ␣0*
Iteration 5	pc_5	*or, true, #*
Iteration 5	pc_6	*or, 0*
Iteration 5	pc_7	*or, "*
Iteration 5	pc_8	*or, /**/, #, ␣, 0, !*
Iteration 5	pc_9	*or, /**/, 1, 1*

To conclude, from the example, we see that having more path conditions, either through more iterations or using RandomForest, helps derive a better understanding of the patterns shared by bypassing attacks. In turn, this puts a firewall administrator in a better position to devise an effective patch for a WAF's rule set.

6.2.6 Discussion

6.2.6.1 Differences between Case Studies

When comparing the testing results of the two case studies some notable differences stand out. First, bypassing attacks could be found for each parameter protected by ModSecurity, whereas with the proprietary WAF, only 29 out of 75 parameters lead to bypassing attacks. This can be attributed to the fact that the latter strictly validates each input to follow an expected format. For example, a value provided to the parameter *credit card number* must consist of 16 to 19 digits and, otherwise, the request is rejected. SQL injection attacks typically require a larger character set and thus all attacks

are blocked. Similarly, for the 46 parameters for which no attack is found, the expected input format prevents attacks. However, it is not possible to define such strict validation rules for all parameters, since the inputs might vary significantly in terms of character set and length. This is the case for the other 29 parameters where the expected input format is very general and the input validation rules rather loose, thus being prone to attacks. For example, the vulnerable parameter *Address* is expected to be a string with a maximum of 35 characters, a constraint with which many of the SQLi attacks comply.

Another major difference between the two case studies are the number of bypassing attacks per tested parameter. For ModSecurity, about 1.000 bypassing attacks per parameter are found while, for the proprietary WAF, they are on average 10.000. This significant difference can be attributed to the attack detection capabilities of each respective firewall and highlights the difficulty of customising a rule set for a particular IT environment in practice. In our experiment, we use a default rule set for ModSecurity, while the proprietary WAF has a customised rule set to match a particular IT system. Such a customisation is often necessary to achieve an acceptable false positive rate, but comes at the cost of reduced attack detection capabilities due in part to the lack of suitable tools to test the firewalls.

6.2.6.2 Application of the Proposed Techniques

We have proposed and evaluated three variants of a machine learning driven technique for the generation of SQLi attacks, namely ML-Driven E, ML-Driven D, and ML-Driven B. ML-Driven D and ML-Driven B entail different strategies in allocating the test generation budget. ML-Driven E reconciles these differences and delivers a better performance. We have compared all these variants with RAN, the baseline technique considered in our work, on ModSecurity (a popular open-source WAF) and a proprietary WAF. Our experiments show that ML-Driven E outperforms all other techniques.

We have also demonstrated the usefulness of mining more bypassing attacks in devising string patterns to fix WAFs. In our context, we experimented with RandomTree and RandomForest as machine classifiers for ML-Driven E. Since we show that the latter helps extract more path conditions that are useful in identifying patterns and fixing WAFs, we recommend the use of RandomForest.

6.3 Related Work

Previous research on testing attack detection capabilities has focused on testing an application's build-in input validation mechanisms as well as testing of dedicated firewalls. In what follows, we review the related work in both areas.

Offutt et al. introduced the concept of Bypass Testing, in which an application's input validation is tested for robustness and security [Offutt et al., 2004]. Tests are generated to intentionally violate client-side input checks and are then sent to the server application to test whether the input constraints are adequately evaluated. Liu et al. proposed an automated approach to recover an input validation model from program source code and formulated two coverage criteria for testing input validation based on the model [Liu and Kuan Tan, 2008]. Desmet et al. verify a given combination of a WAF and a web application for broken access control vulnerabilities, e.g. forceful browsing, by explicitly specifying the interactions of application components on the source code level and by applying static and dynamic verification to enforce only legal state transitions [Desmet et al., 2006]. In contrast, we

propose a black box technique that does not require access to or modification of the tested application's source code. In our approach, we use machine learning to identify the patterns recognised by a WAF as SQLi attacks and generate bypassing test cases that avoid those patterns.

The topic of testing network firewalls is addressed by an abundant literature. Although network firewalls operate on a lower layer than application firewalls, which are our focus, they share some commonalities. Both use policies to decide which traffic is allowed to pass or should be rejected. Therefore, testing approaches to find flaws in network firewall policies might also be applicable to web application firewall policies. Bruckner et al. proposed a model-based testing approach which transforms a firewall policy into a normal form [bru, 2010]. Based on case studies they found that this policy transformation increases the efficiency of test case generation by at least two orders of magnitude. Hwang et al. defined structural coverage criteria of policies under test and developed a test generation technique based on constraint solving that tries to maximise structural coverage [Hwang et al., 2008]. Other research has focused on testing the firewalls implementation instead of policies. Al-Shaer et al. developed a framework to automatically test if a policy is correctly enforced by a firewall [Al-Shaer et al., 2009]. Therefore, the framework generates a set of policies as well as test traffic and checks whether the firewall handles the generated traffic correctly according to the generated policy. Some authors have proposed specification-based firewall testing. Jürjens et al. proposed to formally model the tested firewall and to automatically derive test cases from the formal specification [Jürjens and Wimmel, 2001]. Senn et al. proposed a formal language for specifying security policies and automatically generate test cases from formal policies to test the firewall [Senn et al., 2005]. In contrast, in addition to targeting application firewalls, our approach does not rely in any models of security policies or the firewall under test, such formal models are rarely available in practice.

6.4 Summary

WAFs play an important role to protect online systems. The fast pace at which new kinds of attacks appear and their increasing sophistication require WAFs to be tested and updated regularly as otherwise they will be circumvented. We propose a machine learning-driven approach to test the attack detection capabilities of WAFs. The approach automatically generates a diverse set of attacks, sends them to a WAF under test, and checks if they are correctly identified. By incrementally learning from the tests that are blocked or bypassing the firewall, our approach selects tests that exhibit string patterns associated with bypassing the firewall and mutates them using an attack grammar designed to generate new and hopefully successful attacks. Identified bypassing attacks can be used to learn path conditions, which embody successful attack patterns.

With such a set of bypassing attacks and path conditions that characterize them, a security expert can fix or fine-tune the WAF rules in order to block imminent SQLi attacks. In the attacker-defender war, time is vital. Being able to learn and anticipate more attacks that can circumvent a firewall in a timely manner is very important to secure business data and services.

Though in this chapter our approach is proposed and evaluated in the context of SQL injection attacks, it can be adapted to other forms of attacks by making use of other attack grammars targeting different types of vulnerabilities.

Our key contributions in this work include (i) enhancing our previous proposed techniques by consolidating them and improving their performance, (ii) assessing the influence of two different and adequate machine learning classifiers, and (iii) carrying out a large-scale evaluation on two popular WAFs. Evaluation results suggest that the performance of our proposed testing approach is effective at generating many undetected attacks and provides a good basis to identify attack patterns to protect against. The next chapter proposes an automated approach that transforms the learned attack patterns into a patch for a WAF under test. The goal of the devised patch is to prevent further attacks containing the same patterns without blocking any legitimate requests.

Chapter 7

An Automated Approach to Repairing Web Application Firewalls

The previous chapter introduced an automated testing approach for WAFs that is efficient at finding bypassing attacks. Based on these bypassing attacks, this chapter introduces an approach that improves the attack detection capabilities of a WAF under test. The proposed approach transforms the bypassing attacks and related path conditions into a filter rule that, when added to the WAF's rule set, identifies the bypassing attacks without disrupting legitimate usage of the system. Experimental results show that the generated filter rule is effective at blocking the previously bypassing attacks (*recall* between 49% and 96%), while inducing a small number of false positives (*false positive rate* between 0% and 3%) and, thus, is effective at repairing a WAF.

7.1 The WAF Fixing Problem

Consider the task of inferring some WAF filter rule, i.e. a regex, from a set of known bypassing attacks. The goal of such a regex is to match bypassing attacks without matching any legitimate requests. In this section, we precisely define the problem of finding such a regex and recast it as a combinatorial optimization problem.

There is a potentially infinite space of regex candidates to consider when searching such a regex. The path conditions, which are an output of the WAF testing approach described in Chapter 6, can be used to focus the search to a subset that is likely to contain "good" solution candidates. A path condition describes a string pattern that a group of bypassing attacks have in common. This makes a path condition a natural fit to be translated into a regex in order to match all attacks described by the path condition (see Table 6.6 for an example). Recall that our WAF testing approach typically abstracts several path conditions from the bypassing attacks and each path condition describes a string pattern shared by a distinct set of attacks. In order to infer a regex that matches all identified bypassing attacks, we re-express the problem as a search problem focusing on the space defined by path conditions:

7.1 Definition: Search Space. *Let $P = \{p_1, \ldots, p_n\}$ be a set of path conditions and let $S_i = \{s_{i1}, \ldots, s_{im}\}$ be the set of slices that appear as terms in some path condition $p_i = s_{i1} \wedge \ldots \wedge s_{im}$, $p_i \in P$. The search space $\mathcal{S}$ for the WAF fixing problem is defined by the Cartesian product:*

$$\mathcal{S} = \mathcal{P}(S_1) \times \ldots \times \mathcal{P}(S_n), \tag{7.1}$$

where $\mathcal{P}(S_i)$ denotes the power set of S_i.

According to Definition 7.1, a candidate solution in the search space $\mathcal{S}$ is a n-tuple, which has one element per path condition $p \in P$ and each element in the tuple represents a combination of slices of the corresponding path condition. For example, consider the three path conditions and their corresponding power sets shown in Table 7.1.

Table 7.1. An example of three path conditions and their power sets.

Path Condition	Slices	$\mathcal{P}(S_i)$
p_1	$\{s_{11}, s_{12}\}$	$\emptyset, \{s_{11}\}, \{s_{12}\}, \{s_{11}, s_{12}\}$
p_2	$\{s_{21}\}$	$\emptyset, \{s_{21}\}$
p_3	$\{s_{31}, s_{32}, s_{33}\}$	$\emptyset, \{s_{31}\}, \{s_{32}\}, \{s_{33}\}, \{s_{31}, s_{32}\}, \{s_{31}, s_{33}\}, \{s_{32}, s_{33}\}, \{s_{31}, s_{32}, s_{33}\}$

A candidate solution for the given example is a 3-tuple, e.g. $c_1 = (\{s_{11}, s_{12}\}, \{s_{21}\}, \{s_{33}\})$. The first element of the tuple represents a combination of slices of p_1, the second element a combination of slices of p_2, and the third element a combination of slices p_3. Recall that the slices of a path condition describe a string pattern that a group of bypassing attacks have in common and, hence, by translating the slices of a path condition into a regex the bypassing attacks can be identified and blocked. In the given example, the first element of c_1 can be translated into regex r_0, which identifies any attacks that contain the slices s_{11} and s_{12}. Similarly, the second and third element can be translated into regex r_1 and r_2 that identify attacks containing the slice s_{21} and s_{33}, respectively. Note that the resulting regexes r_0, r_1, and r_2 each match a distinct set of attacks and in order to block all attacks, a superseding regex of the form $r = r_0 \mid r_1 \mid r_2$ is necessary. In other words, r matches the attacks that are matched by either r_0, r_1, or r_2.

A typical test run of the WAF testing approach presented in Chapter 6 yields on average between 50 and 200 path conditions (see Figure 6.12). This number of path conditions results in a potentially very large search space. To systematically guide the search towards an optimal solution, i.e. a regex with a high attack detection rate and low *false positive rate*, a mechanism to assess the quality of a solution is required. In what follows, we define how a solution's quality is assessed.

Let R denote a set of requests that are processed by a WAF such that $A \subset R$ is a subset of malicious requests and $L \subset R$ is a subset of benign requests. Note that each request is either malicious or benign and, thus, $A \cap L = \emptyset$ and $A \cup L = R$. Let $M(c,R)$ denote a set of matched requests by applying a regex c to R, let $M_{tp}(c,R) = \{x \mid x \in A \wedge x \in M(c,R)\}$ denote a set of true positive matches and $M_{fp}(c,R) = \{x \mid x \in L \wedge x \in M(c,R)\}$ a set of false positive matches.

To assess the quality of a candidate solution we use the well-known measures *false positive rate* and *recall*:

7.2 Definition: Objective Functions. *Given a candidate solution $c \in \mathcal{S}$ and a set R of request, we assess the quality of c with:*

$$fpr(c,R) = \frac{\left|M_{fp}(c,R)\right|}{|L|} \tag{7.2}$$

$$recall(c,R) = \frac{|M_{tp}(c,R)|}{|A|} \tag{7.3}$$

A solution candidate is evaluated with respect to both objective functions, i.e. we search a solution that maximizes *recall* and minimizes the *false positive rate*. The WAF Fixing Problem consists of finding optimal trade-off solutions between *recall* and *false positive rate*. Note that there might be several such optimal trade-off solutions that form a Pareto front. To compare the quality of two candidate solutions, we define a relation $\prec$ that is based on the objective functions:

7.3 Definition: Dominated and nondominated Solutions. *Given a candidate solution $c \in \mathcal{S}$ and a set R of request, c is said to be* ***nondominated*** *if:*

$$\begin{aligned} \nexists c' \in \mathcal{S} : fpr(c,R) > fpr(c',R) \wedge recall(c,R) \leq recall(c',R) \quad \vee \\ fpr(c,R) \geq fpr(c',R) \wedge recall(c,R) < recall(c',R) \end{aligned} \tag{7.4}$$

Accordingly, c is said to be ***dominated*** *if there is a $c' \in \mathcal{S}$ that satisfies Equation 7.4. In this case, we use the notation $c \prec c'$ to indicate that c' dominates c.*

In our problem context, the practical implications of a legitimate request being wrongly classified as an attack, i.e. a false positive, are more severe than that of an attack wrongly classified as a legitimate request, i.e. a false negative. Before accepting a solution candidate, the user has to decide for each false positive caused by the solution candidate if it impacts the functionality of the system under test and, if so, choose the next best solution candidate. Therefore, we introduce a constraint on the number of false positives to limit the required manual effort:

7.4 Definition: Feasible Candidate Solution. *We say $c \in \mathcal{S}$ is a feasible candidate solution if it satisfies the constraint:*

$$\left|M_{fp}(c,R)\right| < t\,, \tag{7.5}$$

where t is a user-defined threshold for an acceptable number of false positives. Additionally, $\mathcal{X} \subseteq \mathcal{S}$ denotes the set of all feasible candidate solutions.
Based on the constraint we define a function g that measures the degree of constraint violation:

$$g(c) = \begin{cases} 0 & \text{if } c \in \mathcal{X} \\ \left|M_{fp}(c,R)\right| - t & \text{if } c \notin \mathcal{X} \end{cases} \tag{7.6}$$

The problem addressed in this chapter can now formally be defined as the optimization problem:

7.5 Definition: The WAF Fixing Problem. *Given a set $L \subset R$ of benign requests, a set $A \subset R$ of malicious requests, a set of path conditions defining the search space $\mathcal{S}$, and a set $\mathcal{X} \subseteq \mathcal{S}$ of feasible candidate solutions, then the WAF Fixing Problem is to compute a set $\mathcal{C}_{opt}$:*

$$\mathcal{C}_{opt} = \{c \mid c \in \mathcal{X} \wedge \nexists c' \in \mathcal{X} : c \prec c'\} \tag{7.7}$$

According to Definition 7.5, the WAF Fixing Problem consists of finding a set C_{opt} of nondominated solutions in the space of feasible solutions.

7.2 Approach

This section introduces an approach to solve the WAF Fixing Problem. The presented approach is based on the genetic algorithm NSGA-II [Deb et al., 2002]. We selected NSGA-II because it is applicable to multi-objective optimization problems with a constrained search space like the WAF Fixing Problem. The authors evaluate NSGA-II on several optimisation problems, of which many have two objectives like the WAF Fixing Problem, and find that NSGA-II performs the best amongst the compared genetic algorithms. We used the implementation of NSGA-II in JMetal, a popular framework for genetic algorithms [Nebro et al., 2015].

A Genetic Algorithm (GA) is a search heuristic that is inspired by the evolutionary process in nature, where the fittest individuals of a population prevail and pass on their genes to their offspring. A GA imitates this concept and evolves a population of candidate solutions iteratively with the goal of breeding better candidate solutions over multiple iterations. Starting from a randomly generated initial population of candidate solutions, a GA generates an offspring population by performing crossover and mutation operations on the individuals and by selecting the best candidate solutions according to some criterion. The selection of candidate solutions is typically performed based on a fitness value, which reflects the degree with which a candidate solution achieves a desired objective. In the following, we describe how the essential components of a genetic algorithm, i.e. chromosomes, selection, mutation, and crossover, are implemented in our context.

Chromosomes. A fundamental decision when using GAs is how to encode a candidate solution and its properties. In the terminology of GAs, a solution candidate is called a *chromosome*. A chromosome is composed out of several *genes*, each representing a possible value of a variable of the candidate solution. In our context, each path condition that is used to define the search space is a gene of the chromosome and each slice that is part of a path condition is encoded as a bit in the gene.

Table 7.2. An example of two candidate solutions encoded as chromosomes based on the path conditions and slices depicted in Table 7.1.

id	Candidate Solution
c_1	$(\{s_{11}, s_{12}\}, \{s_{21}\}, \{s_{33}\})$
c_2	$(\{s_{12}\}, \emptyset, \{s_{31}, s_{33}\})$

Table 7.3. Candidate Solutions

p_1		p_2	p_3		
s_{11}	s_{12}	s_{21}	s_{31}	s_{32}	s_{33}
1	1	1	0	0	1
0	1	0	1	0	1

Table 7.4. Corresponding chromosome encoding.

Table 7.2 shows an example of two candidate solutions and their corresponding encoding as chromosomes. The path conditions and slices used in this example are from the previous example in Table 7.1. As in the previous example, there are in total three path conditions and, hence, the candidate solutions c_1 and c_2 are 3-tuples. The first element of c_1 and c_2 represents a combination of slices of path condition p_1. For c_1, the first element is $\{s_{11}, s_{12}\}$ and, thus, the chromosome encoding of p_1 is $s_{11} = 1$ and $s_{12} = 1$. For c_2, the first element is $\{s_{12}\}$ and, thus, the chromosome encoding of p_1 is $s_{11} = 0$ and $s_{12} = 1$. The second and third element of c_1 and c_2 are encoded in the same fashion.

Crossover and Mutation. To efficiently sample the search space, a GA creates new solution candidates by performing crossovers and mutating the chromosomes of existing solution candidates. To perform a crossover, two candidate solutions are randomly selected from the population and their chromosomes are swapped starting from a randomly chosen gene in the chromosome. In the literature, this method is called single-point crossover [Mitchell, 1998]. In contrast to crossover, which recombines existing genes, the purpose of mutation is to introduce new genes into the population. In our context, the mutation operation is implemented by flipping a randomly selected bit of a chromosome.

Selection. To effectively guide the breeding of offspring generations towards a desired objective, a GA chooses the fittest individuals in a population and includes them in the offspring generation. NSGA-II prioritises nondominated individuals in its selection process, a concept known as *elitism*, and favours diverse individuals using a crowded distance estimation. Maintaining a population of diverse individuals is important to prevent that the search prematurely converges towards a single area. Therefore, the crowded distance estimation in NSGA-II tries to select a uniformly spread-out Pareto front.

To perform the selection process, NSGA-II has to compute the fitness of the candidate solutions in the population. In the context of the WAF Fixing problem, we measure the fitness of a candidate solution with the objective functions *false positive rate* and *recall*, as defined in Definition 7.2.

7.3 Evaluation

In this section, we apply the approach introduced in the previous section to the WAF Fixing Problem and evaluate the quality of the found solutions. The approach is applied to three datasets in total: Two datasets from a WAF of a financial institution that processes thousand of transactions daily and one dataset from a popular open-source WAF.

7.3.1 Research Questions

We address the following research questions in our evaluation:

RQ1: *How effective are the found regular expressions in identifying bypassing attacks?*

RQ2: *To which extent do the found regular expressions misclassify legitimate traffic as attacks?*

The goal of the proposed approach is to find a regular expression that identifies a set of bypassing

malicious requests without affecting legitimate requests. Therefore, **RQ1** investigates how many malicious requests a generated regex identifies, i.e. *recall*. **RQ2** investigates if the regex is misclassifying legitimate requests as attacks, i.e. *false positive rate*.

7.3.2 Subject Applications

To answer the research questions we designed two sets of experiments. The subject applications used in both settings are identical to the ones used in the evaluation of our WAF testing strategy in Chapter 6 (for a description of the experimental settings and the subject applications refer to Section 6.2.1). In brief, the first set of experiments uses popular open-source subject applications, ModSecurity and Cyclos, while the second setting uses proprietary software of a financial institution that processes thousands of requests daily.

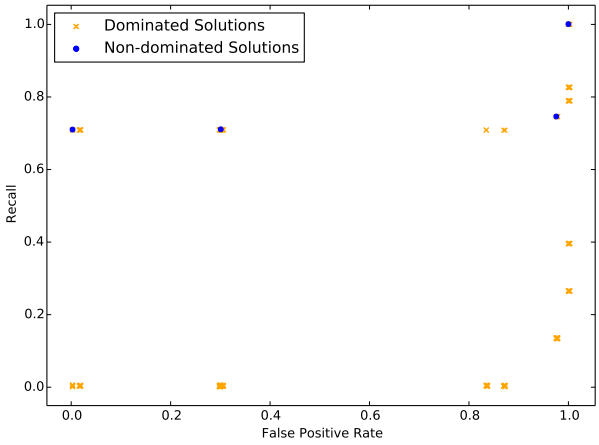
To evaluate our proposed approach we collected for each WAF under test a set of malicious requests that are not correctly identified by the WAF and a sample of benign requests. The benign requests represent the legitimate usage of the web application and, thus, the WAF under test is expected to let these requests pass. We collected the benign requests by executing the functional test suite of the web application and logged each request that was sent to the web application. The malicious requests were obtained by executing our WAF testing strategy and the path conditions were learned from the malicious requests as detailed in Chapter 6. Table 7.5 lists the total number of requests that were collected for both experimental settings.

Table 7.5. Experimental Settings.

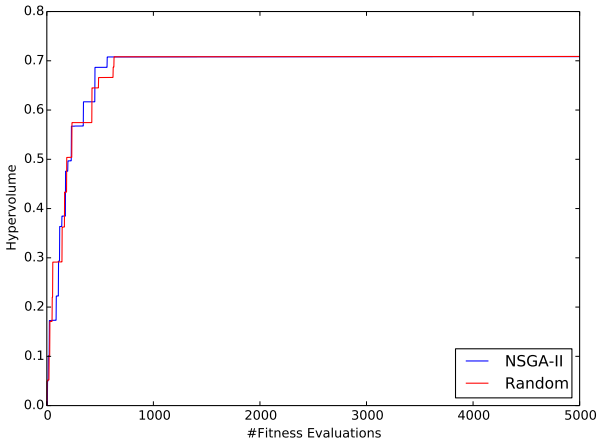
Setting	Operation	#Benign Request	#Attacks	#Path Conditions
Open-Source	doPayment	369	1244	39
Industrial	Operation 1	95	943	49
	Operation 2	299	14385	102

7.3.3 Results

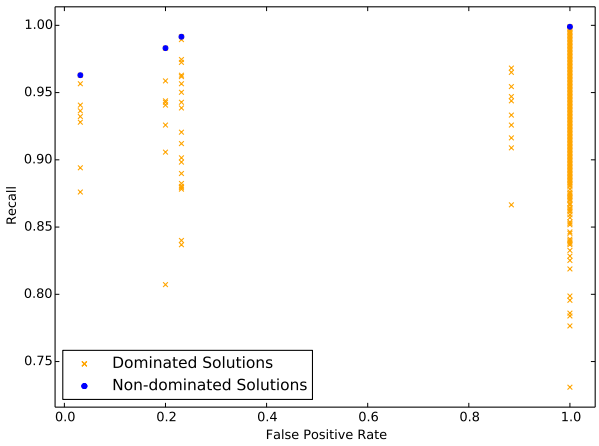
This section presents the results of the performed experiments. We conducted three experiments in total, one for each of the datasets introduced in the previous section. In each experiment, we applied the genetic algorithm to the WAF Fixing Problem using the legitimate requests, attacks, and path conditions of the corresponding datasets. The genetic algorithm is configured to generate 200 generations. When generating offspring populations, the probability that a candidate solution is mutated is set to 0.1 and the crossover probability is set to 0.9. The probabilities are chosen according to recommendations in the literature [McMinn, 2004, Mitchell, 1998].



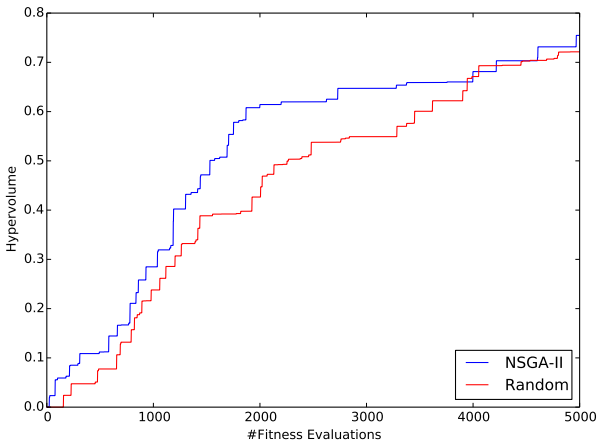
(a) Operation *doPayment*.



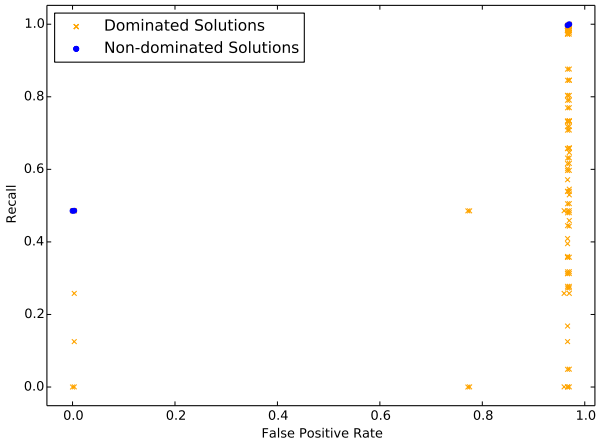
(b) Operation *doPayment*.



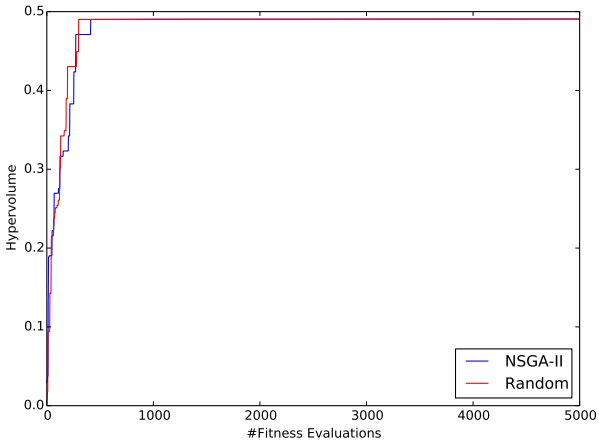
(c) Operation *Operation 1*.



(d) Operation *Operation 1*.



(e) Operation *Operation 2*.



(f) Operation *Operation 2*.

Figure 7.1. Scatter plot of the solutions found by NSGA-II (left) and a comparison between NSGA-II and random search depicting the increase of the hypervolume over the number of fitness evaluations (right).

The plots depicted on the left side of Figure 7.1 show a scatter plot of the found solutions for each experiment. The solutions are compared based on how many legitimate requests a regex misclassifies as attack (*false positive rate*) and how many attacks are correctly identified (*recall*). *False positive rate* and *recall* are calculated according to Definition 7.2. Furthermore, we divide the found solutions into dominated and nondominated solutions according to Definition 7.3.

Table 7.6 shows the nondominated solutions for each dataset depicted in Figure 7.1. In all experiments, the genetic algorithm finds solutions with either zero false positives (*doPayment*, *Operation 2*) or with a low *false positive rate* of 3% (*Operation 1*) and a high *recall* (71% *doPayment*, 96% *Operation 1*, 49% *Operation 2*). In addition, for all datasets there are alternative nondominated solutions with a higher *recall* but also a higher *false positive rate*. Such alternative solutions might be preferable if the resulting false positives can be tolerated or if the legitimate request causing the false positive can be white-listed.

Table 7.6. Overview of the nondominated solution per dataset.

Operation	Non-dominated Solutions	
	FPR	*Recall*
doPayment	0	0.71
	0.3	0.711
	0.98	0.75
	1	1
Operation 1	0.03	0.96
	0.2	0.98
	0.23	0.991
	1	0.998
Operation 2	0	0.49
	0.003	0.494
	0.966	0.997
	0.969	1

We compare NSGA-II to a random search approach to assess the quality of the nondominated solutions found by NSGA-II. The comparison is based on the well-known quality indicator hypervolume [Zitzler and Thiele, 1999]. A hypervolume measures the volume in the solution space that is covered by members of the set of nondominated solutions. The larger the volume, the better the solutions found by the approach. We executed both approaches, NSGA-II and random search, and measured each time a nondominated solution was found the hypervolume. The plots depicted on the right side of Figure 7.1 show the hypervolume for each approach and dataset. For the dataset corresponding to *Operation 1*, the hypervolume of the solutions found by NSGA-II increases faster compared to random search; hence NSGA-II requires less time to find good solutions. For the other two datasets, the increase in hypervolume for NSGA-II and random search is comparable and its fast growth indicate that good solutions are easy to find. In consequence, choosing NSGA-II over random search is reasonable since it performs better when good solutions are hard to find (*Operation 1*) and comparable otherwise (*doPayment*, *Operation 2*).

Based on the presented results we can answer **RQ1** and **RQ2**:

The attack detection rate of the nondominated regular expressions ranges from 49% to 96% depending on the dataset.

The false positive rate of the nondominated regular expressions ranges from 0% to 3% depending on the dataset.

To conclude, in our conducted experiments the proposed approach is effective at finding regular expressions that do not misclassify legitimate requests, or only to a very limited degree, and that identify a considerable number of attacks. Given that the proposed approach is automated, the attack detection capabilities of the WAF under test can be significantly increased at the cost of only computation time. If complete automation is not a requirement, an IT Security Engineer can choose between alternative regular expressions that provide varying degrees of *recall* and *false positive rate*.

7.4 Summary

WAFs are an important component in corporate IT environments and play a vital role in protecting potentially vulnerable web applications and services. However, the rule sets of firewalls tend to become complex and their configuration and maintenance can become time consuming and error-prone, which might result in attacks staying undetected [Wool, 2010, Wool, 2004]. Therefore, WAFs should be tested regularly and their rule set should be adjusted to avoid that attacks are bypassing the WAF.

This chapter proposes an automated approach to increase the attack detection rate of a WAF. We formalised the WAF fixing problem as a combinatorial optimisation problem and presented an approach, which is based on the genetic algorithm NSGA-II, to solve the problem. Given a set of bypassing attacks and corresponding path conditions, which are the test output of the WAF testing strategy presented in Chapter 6, our approach infers a regular expression that, when added to the WAF's rule set, prevents the attacks from bypassing. By providing our approach with a set of legitimate requests that represent the benign usage of the application protected by the WAF, we can guide the genetic algorithm towards regular expressions that do not match legitimate requests and do not affect the benign usage of the application. Experimental results show that the generated filter rule is effective at blocking the previously bypassing attacks (*recall* between 49% and 96%), while inducing a small number of false positives (*false positive rate* between 0% and 3%) and, thus, is effective at repairing a WAF.

Chapter 8

SOFIA: An Automated Security Oracle for black box Testing of SQL-Injection Vulnerabilities

When engineering secure software systems and services, software testing is one of the main practices to detect faults as well as security vulnerabilities. Security testing (also called penetration testing) is a branch of software testing devoted to stress programs with respect to their security features, with the aim of identifying vulnerabilities. Security testing involves two major challenges, generating input values (referred to as *test payloads*), intended to exercise vulnerabilities, and evaluating whether such payloads manage to expose an actual vulnerability. The security oracle addresses the latter.

Security testing is highly expensive given the complexity of modern systems, typically providing a wide range of services, and the sophistication of attacks and exploitations. To reduce effort and cost, the research community has focused on automating security testing. Regarding SQLi, the test input generation problem has been extensively investigated and automated approaches are available [Appelt et al., 2014, Halfond et al., 2006a, Kieyzun et al., 2009a, Kindy and Pathan, 2011]. Automating the test oracle problem for SQLi vulnerabilities, however, remains an open problem. This is a significant obstacle to test automation, as manual oracles severely limit the number of test execution results a test team can process [Barr et al., 2015].

In this chapter, we present SOFIA, a **S**ecurity **O**racle for SQL**i** **A**ttacks. Our goal is to satisfy three important requirements. First, it must be independent from known attack instances so that new types of attacks can be detected in the future. This is an important leap forward since existing solutions based on attack patterns can only detect publicly known and documented attacks. Second, the oracles should not rely on knowledge about test input data or their generation algorithm in order to be usable with any given test generation tool. Third, our proposed oracle should not require the source code of the SUT, since we target black box testing. This is often a mandatory requirement for external security testing (carried out by third-party penetration testers) or for systems whose source code is not available.

Most of the existing SQLi oracles either require known attacks in the learning phase [Pinzón et al., 2013] or access to source code [Halfond and Orso, 2005a, Bisht et al., 2010, Buehrer et al., 2005, Kemalis and Tzouramanis, 2008]. The few approaches that still meet all the three requirements [Liu

et al., 2009, Valeur et al., 2005] are fundamentally different than our solution in ways that affect recall and false positive rates. Whereas they detect user inputs in SQL statements and compare them with user inputs observed at learning time, we prune data from SQL statements and compare their parse trees. By comparing structure instead of data, our goal is to enable SOFIA to yield high recall and low false positive rates. The main motivation is that modelling all possible safe data is highly difficult, if feasible at all, and incomplete models cause false positives. Further, we observed that a change in query structure is the most direct manifestation of an SQLi attack.

SOFIA is built using one-class machine classification. SQL statements issued by a SUT to its database are logged and parsed to create SQL parse trees, which are fed to a clustering algorithm. The *Tree edit distance* is used to measure the distance amongst parse trees. Our approach consists of two phases: *Training* and *Testing*. During training, *legitimate* SQL statements, which are obtained from regular executions, are grouped into clusters of similar statements. We refer to this set of clusters as a *safe model*. Such a model represents legitimate database accesses in the absence of attacks.

In the testing phase, when test inputs trigger new SQL statements, our oracle assesses whether the statements can be assigned to the clusters of the safe model. In the positive case, we can assert that the statements are safe and no vulnerability is reported. Otherwise, such statements are classified as anomalous, and hence, vulnerability alerts are reported. We have carried out an experimental evaluation in terms of false positive rate, recall rate, and computational cost on six real applications and with three different attack generation tools. The obtained results show that the proposed oracle achieves a very low false positive rate (0.6%) and misses no attack (100% recall) with a low computational overhead.

The proposed oracle is meant to support security testing, by classifying SQL statements triggered by test cases as legitimate statements or as SQLi attacks. However, since it relies on a black box strategy and is trained only on legitimate executions, it could be also deployed as a database firewall in production to filter SQL statements and block SQLi attacks before they are actually executed. Investigating such potential application is out of the scope of this chapter though and we present and assess the proposed oracle only in the context of security testing.

In Section 8.2, we discuss the requirements and our strategy for the security oracle. In Section 8.3 we present in detail our approach. Section 8.4 reports our experiments to assess the accuracy and speed of the oracle. Finally, Section 8.6 summarizes this chapter.

8.1 Background

This section introduces the concept of *tree edit distance*, which is used in our approach to compare the similarity of SQL statements.

Ordered labelled trees refer to a tree structure in which nodes are labelled and edges capture predecessor-successor relationships amongst nodes. The left-to-right order amongst siblings is also significant to the semantics of the trees. Parse trees that structure sentences or programs according to some context-free grammars are ordered labelled trees.

Transforming one ordered labelled tree (or just tree for brevity) into another involves three basic

types of edit operations: *changing a node label*, *delete a node*, and *insert a node*. As an example, taking the trees t_1 and t_2 in Figure 8.1, transforming t_1 into t_2 can be performed with a sequence of four edit operations: *change e to k, delete c, delete d, add h*, or alternatively: *change e to k, change d to h, delete c*.

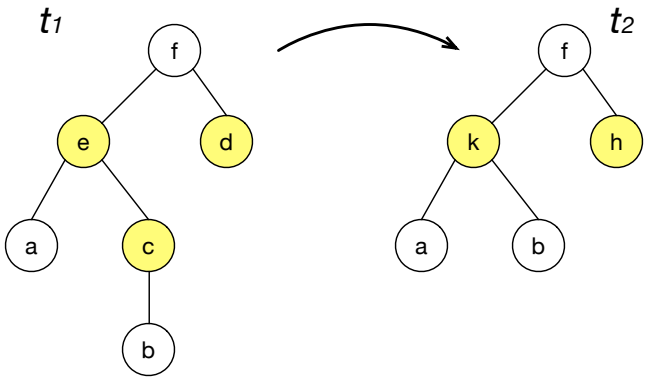


Figure 8.1. An example of two ordered labelled trees.

Formally, each edit operation o_i is assigned a cost (usually one unit). The cost of a sequence of edit operations $S_j = \langle o_1, o_2, \ldots, o_N \rangle$ is the sum of the cost of all operations o_i. Since there are usually alternative sequences to transform a tree into another, the tree edit distance between two trees is the minimum cost amongst possible sequences. When the cost of all edit operations is equal to one, then the edit distance between two trees is the number of operations of the shortest sequence that transforms one tree into the other.

8.2 Requirements and General Strategy

8.2.1 Security Oracle Requirements

Most of the classifiers used as security oracles are learned on a training set that contains both positive and negative examples (attacks and legitimate executions) [Avancini and Ceccato, 2013]. However, these approaches are expected to suffer from two main limitations, namely, (i) the availability of attacks, and (ii) the representativeness of attacks, especially when the sample is small as in most practical contexts.

Documented attacks are usually unavailable for many systems. Some ethical attackers make their techniques and payloads available on the Internet, but software developers are usually unaware of them. Even if they are, the number of known attacks is limited. We also need to account for unknown, new attacks that may appear after the system's deployment. As a result, a security oracle would be much more beneficial for testing (and also in other contexts like monitoring in production) if it does not require the availability of documented attacks.

In addition, even when available, attacks used to train security oracle classifiers are often expected to be *representative* of possible attacks that could target a system. Unfortunately, this is normally not the case as new attacks are being introduced at a very fast pace. Therefore, a classifier should not rely on documented attacks at the risk of being ineffective with new ones. Thus, the first requirement for a security oracle is the following:

Requirement Req$_1$*: The security oracle should be independent from known instances of successful attacks.*

Often in security testing, oracles depend on knowledge about what attack generation algorithm and what test input data have been used to determine the expected output if there is a vulnerability. For example, some oracles classify a test as a successful attack when the execution output contains the same attack payloads as the inputs [Kieyzun et al., 2009a]. This strategy suffers from the observability problem as the output can be masked by a generic error message, or worse, the impact of the successful attack cannot be easily observed by the tester, e.g., like in *second order* SQLi attacks. Furthermore, this strategy, because it is specific to test generation algorithms or input data, limits the portability of the oracle. As a result, it may not work with most attack generation tools, without additional adaptation overhead. It is desirable to define an oracle that is independent from attack generation strategies so that it can be used with many attack tools and a variety of attack generation approaches. The second requirement is, thus:

Requirement Req$_2$*: The security oracle has no knowledge on what input data are used to test the system.*

Often, systems are written using frameworks and third part libraries. Since commercial libraries are rarely distributed with source code, a whitebox approach in the generation of a security oracle is of limited applicability in real industrial settings. In many contexts, such as for third-party penetration testers, access to source code is not an option. Thus, the third requirement is:

Requirement Req$_3$*: The security oracle should not rely on the source code of the SUT.*

8.2.2 Our Strategy

Our strategy to create a security oracle for SQLi vulnerabilities that satisfies the above requirements relies on a black box, security safe model, which is a model of safe execution inputs.

Safe model: To be independent from known attacks and attack tools, we decided to exclude attacks from the training set used to learn the security oracle. We propose a security oracle that only takes into consideration legitimate executions and builds a model of safe SQL statements. Tests will be classified as legitimate if they generate SQL statements satisfying this safe model and potential attacks otherwise.

Black box: The SQL statements sent by the SUT to the back-end database are the only features considered by the oracle, either in the training phase and in the testing phase. As a result, such an oracle can be deployed to test systems developed in many languages and has no dependency or limitation regarding specific development frameworks.

The proposed safe model depends on the specific legitimate executions that are being considered. However, using classical black box testing techniques, it is much easier to generate a large number of representative legitimate inputs than it is to generate attacks. The next section explains in detail the process to construct our security oracle.

8.3 SOFIA: The Security Oracle

The procedure to build and apply the security oracle is summarized in Figure 8.2. It consists of two phases, *Training* and *Testing* including five steps: *Parsing*, *Pruning*, *Computing Distance*, *Clustering*, and *Classification*. These two phases share the first three steps. *Clustering* is exclusively part of training while *Classification* is exclusively part of testing.

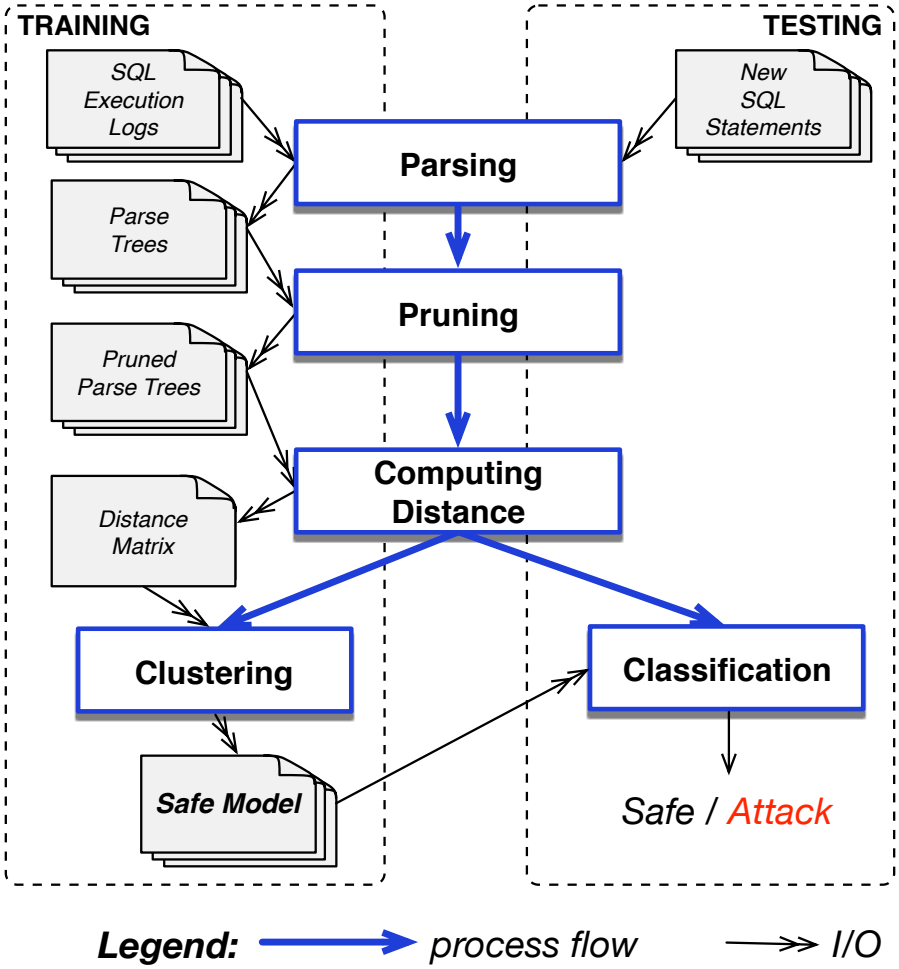


Figure 8.2. The overview SOFIA: the training process for learning safe models from SQL execution logs; the classification process for classifying new SQL statements.

The process starts with a set of SQL statements, obtained from safe executions of a SUT, either by executing functional tests or by monitoring regular system executions. The SQL statements are parsed and the parse trees represent the objects to be classified. The fact that our oracle uses only legitimate statements allows us to avoid the task of manual labelling training data (as legitimate statements or attacks), as often required by other supervised techniques.

Information from the parse trees, which is specific to concrete SQL statements and irrelevant for detecting attacks, is removed by pruning the parse trees. This helps not only in reducing the number of unique trees to be clustered and better scale, but also improves the overall attack detection performance.

Clustering relies on the edit distance amongst pruned parse trees. *Clustering* is used to group together similar SQL statements. Statements with low distance are assigned to the same cluster, while statements with larger distance are assigned to different clusters. The final safe model consists of the optimal partition of SQL statements computed by clustering. Note that the training process that creates safe models from SQL execution logs takes place only once. Safe models are then ready to support security testing in detecting SQLi vulnerabilities.

New statements triggered by executing security tests will be classified using the safe model, by assessing their distance to the centres of the clusters. If a new statement is close enough to a cluster centre, it satisfies the model and is classified as benign statement. Conversely, in the case a new SQL statement does not fit into any existing cluster, it is considered anomalous and classified as a potential attack. Further details of this process are provided in the sections that follow with the help of a running example.

Regarding the defined requirements for the security oracle, Requirement 1 (independence from known attacks) is satisfied since our approach relies only on logs of benign executions. Moreover, the fact that the oracle considers only database logs ensures that requirement 3 is also achieved: no access to the application source code is required. Our classification procedure complies with requirement 2 since it exclusively relies on the SQL statements sent to the database, and not the test case input values.

8.3.1 Training Data

Training data are used to construct the security model. However, differently from other approaches that require both attack and legitimate SQL statements, our approach only relies on the latter to learn a classifier. In fact, our goal is to construct a model of *safe* executions, and classify as *anomalous* everything that does not conform to this model.

Training data are collected by executing the functional test suite of a SUT, or by monitoring its usage during production or acceptance testing. We collect execution logs containing SQL statements executed on the database. To this end, different technologies can be used depending on the underlying DBMS. For *mysql* we use the *mysql-proxy* tool[1] that monitors all traffic to and from *mysql* databases. This solution can be ported to other DBMSs as well. For example, for Java applications that use a JDBC driver to connect to Oracle Database, we can customise the driver to log SQL statements.

Let us use a running example where the execution log includes the SQL statements shown in Figure 8.3. While the three statements query the *user* and *password* columns from the table *users*, they differ from one another in the where conditions.

8.3.2 Parsing

An SQLi attack modifies the semantic of SQL statements, usually by replacing a value with a piece of SQL code, for example by adding a tautology to the *where* clause, or by injecting an additional *select* or *union* statement. These injections, if successful, result in SQL queries that are valid SQL

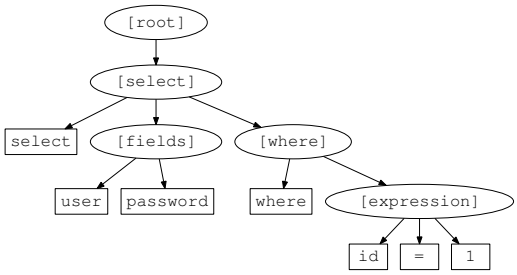
[1] https://dev.mysql.com/downloads/mysql-proxy

```
stmt1: select user, password from users
                                where id = 1;

stmt2: select user, password from users
                                where id = 2;

stmt3: select user, password from users
                                where id = 4 and role = 1;
```

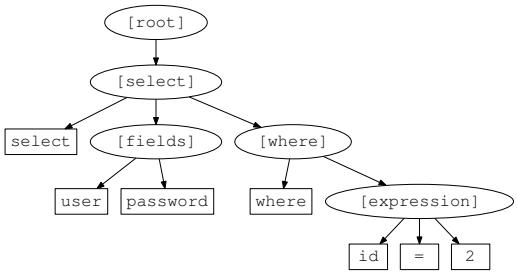
Figure 8.3. Three samples of SQL log of the running example.

statements according to the SQL grammar, but yet have different parse trees. We resort to the parse trees of SQL statements to detect SQLi attacks.

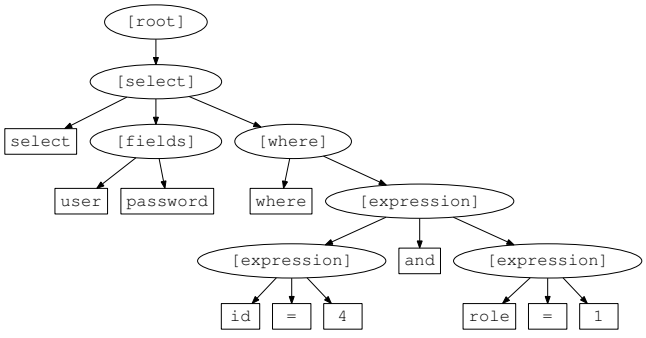
In our work, we rely on the General SQL Parser[2] (GSP for short). GSP is a Java library that supports various DBMSs, including Oracle, SQL Server, DB2, MySQL, Teradata, and Access. Output parse trees are stored as XML documents for other analysis.



(a) stmt1



(b) stmt2



(c) stmt3

Figure 8.4. Parse trees of the SQL statements.

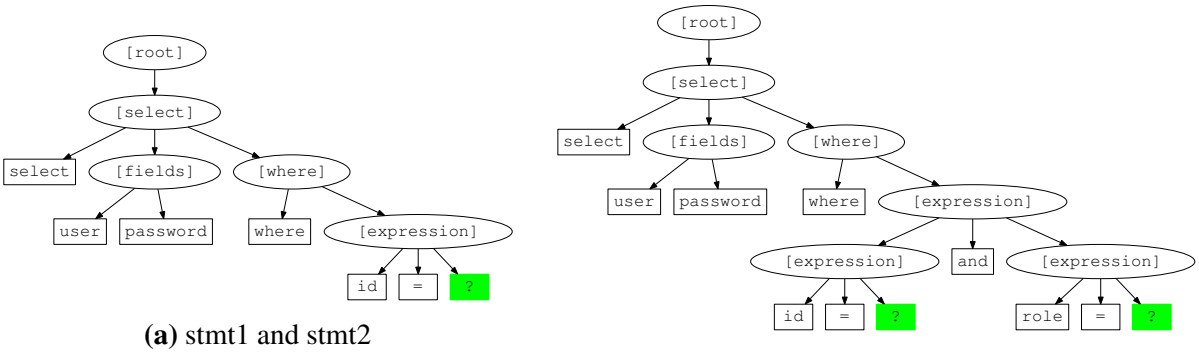
Figure 8.4 shows the parse trees for the SQL statements of our running example. As we can see in the figure, the parse trees contain information that is irrelevant and therefore detrimental to the process of learning a classifier through clustering, e.g., specific user ids. Indeed, some elements in the trees are very specific to the captured SQL executions and are irrelevant for the detection of attacks. The pruning process, described next, aims at removing such irrelevant information from the parse trees.

[2]http://sqlparser.com/

8.3.3 Pruning

Pruning could be done according to different strategies depending on what piece of information should be removed. To decide about the most effective pruning in our context, we should consider how attacks are typically carried out. SQLi attacks aim at altering SQL statements by replacing data with a new piece of SQL code, i.e. a string or numeric literal is replaced by code. Thus, two legitimate SQL statements collected during the execution of the same feature but with different input data should only differ in terms of data values. Instead, a legitimate statement and an attack statement should differ not only with respect to data but also the SQL command structure in the maliciously injected part.

Based on the above consideration, we decided to prune data values in parse trees: We replace all the constant numeric and string values in the tree with the same placeholder (e.g. with the empty string or with the constant zero). As a result, statements that just differ in values are characterised by a single pruned tree.



(a) stmt1 and stmt2

(b) stmt3

Figure 8.5. Pruned parse trees of the example SQL statements.

Figure 8.5 shows the pruned parse trees of our running example. Nodes with data values are replaced with a place holder (the ? character). Since *stmt1* and *stmt2* only differ in data value of the *id* attribute, their pruned versions are equivalent. For learning purposes, redundant versions of the same pruned trees will be discarded. Out of the three statements of the training set for the running example, only two distinct pruned trees will be considered to construct the safe model.

Our pruning procedure is straightforward. We analyse the XML parse trees and replace the content of the leaf nodes of type *Tconstant*, which contain concrete data values, with a placeholder.

8.3.4 Computing Distance

Parse trees of SQL statements are ordered and labelled trees. A metric of tree edit distance for this class of trees has been proposed by Zhang et al. [Shasha and Zhang, 1990], as discussed in Section 8.1. Zhang et al. have also proposed a fast algorithm to calculate tree edit distances in a polynomial time complexity. Our work makes use of a tool called *approxlib*[3], which implements this specific algorithm.

Let us consider the SQL statement of the attack in Figure 8.6(a), for which the corresponding (pruned) parse tree is shown in Figure 8.6(c). We note that this tree is quite similar to the parse

[3]http://www.cosy.sbg.ac.at/ augsten/src

tree of the legitimate statement 3 in Figure 8.5(b). In fact, we note that these trees are more similar (distance is 11) than the two legitimate statements *stmt1* and *stmt3* (distance is 22). This example highlights the fact that parse tree distance alone is not enough to detect attacks. We need to infer a classification amongst legitimate statements, in this case using clustering to group similar trees, and compare a candidate attack with its closest cluster. The underlying rationale is that, though a legitimate SQL statement does not obviously need to be similar to all legitimate statements resulting from test executions, there should be a cluster with similar statements. Once parse tree distances are computed amongst all the pairs of pruned parse trees, an algorithm can be applied to generate an optimal set of clusters, as described next.

8.3.5 Clustering

We address our clustering problem using the k-medoids algorithm [Reynolds et al., 2006]. This algorithm is a variant of k-means clustering, based on the search for k representative samples, namely the *medoids*, amongst the observations of the dataset. After finding a set of k medoids, k clusters are constructed by assigning each observation to the nearest medoid. We need to find k representative samples that minimize the sum of the dissimilarities of the observations to their closest representative object. Furthermore, we consider a measure of cluster *diameter*. The diameter of a cluster is the maximal distance between the observations in the cluster and its medoid. To build a safe model, training data are clustered into k clusters, characterised by their medoids and diameters.

The adoption of k-medoids clustering is more appropriate in our context than the standard k-means [Jain, 2010] approach. K-means involves the notion of *mean* point, which in our case would mean an average tree for all the observations in the same cluster. Since such a hypothetical tree is insensible in our context, we adopt k-medoid, instead. It picks a representative element for each cluster (i.e., the *medoid*), instead of computing a fictitious average tree.

It is very important to identify the appropriate number of clusters k. Clusters should be small enough to distinguish attacks from legitimate statements based on parse tree distances, but clusters should also be large enough to capture representative groups of similar legitimate statements. More specifically, a too small value of k would elaborate a partition that contains few large clusters. A large cluster would contain very different parse trees, with large distances from each other, and its medoid would not be representative of all the members of the clusters. Also, with large clusters, the distance between an attack and the cluster medoid may be comparable to the cluster diameter. Thus, actual attacks would be wrongly classified as legitimate statements (false negatives).

On the other hand, a large number of clusters k may result in many clusters that contain too few elements to be representative and enable reliable comparisons with new parse trees. False alarms (false positives) may result from new legitimate statements whose tree is at a distance from the closest medoid that is higher than the cluster's diameter.

To decide the most appropriate number of clusters (i.e., the value of k), we adopt a standard approach, the Akaike information criterion (AIC) [Aggarwal and Reddy, 2013, Manning et al., 2008]. It entails balancing the trade-off between the goodness of fit of the model and the size of the model. The underlying rationale is to increase the complexity of the model (i.e., k increases) as far as the gain in precision is high, and stop when the increase in precision is not significant. To achieve this objective, we adopt a penalty factor for each new cluster. To determine the number of clusters in this

way, we select the best k that minimizes the fitness function f composed of two terms: (i) *distortion*, a measure of the extent to which SQL statements deviate from the prototype of their clusters (e.g., *RSS* for k-medoids); and (ii) the *model complexity* that is proportional to the number of clusters:

$$f(k) = RSS(k) + 2 \cdot M \cdot k\,, \tag{8.1}$$

where *RSS* is the residue sum of square, i.e. the error that we commit by approximating each observation in a cluster by the corresponding medoid, and M is the dimensionality of the vector space. In our case M=1, because the only feature used in clustering is the tree-edit distance.

The resulting optimal set of clusters, each associated with a medoid and diameter, represent the safe model used as an oracle to classify newly executed SQL statements. Table 8.1 shows the final clustering configuration for the running example.

Table 8.1. Clustering results for the running example.

Cluster	Medoid	Elements
1	*stmt1*	*stmt1*, *stmt2*
2	*stmt3*	*stmt3*

8.3.6 Classification

Intercepted SQL statements undergo the testing (classifying) process depicted in Figure 8.2: they are parsed, pruned, and eventually classified as safe or malicious. The classification procedure is described with respect to the test sample (a malicious statement) shown in Figure 8.6 and consists of the following steps:

1. Parsing and Pruning: When the test SQL statement (Figure 8.6b) is intercepted by our database proxy, it is parsed (Figure 8.6c, malicious injected code highlighted in red) and pruned (Figure 8.6d, pruning highlighted in green) as described previously.

2. Computing Distance: The tree edit distance is used to identify the closest cluster in the safe model. To compute this result, the pruned parse tree T of the test (Figure 8.6d) is compared to the pruned parse tree of each medoid. In our example, the distances from the medoids are shown in Figure 8.6a as 18 and 11, respectively.

The medoid with the smallest distance from the test (*stmt3* in our example) determines the nearest cluster (cluster 2). Determining the nearest cluster is fast since the medoids' parse trees are pre-computed and only k tree edit distances need to be evaluated. In the example only two comparisons are required.

3. Distance-versus-diameter classification: We check whether the test fits the nearest cluster by comparing its diameter and the distance between the test parse tree and medoid. Alternative measures could be used, such as the distance corresponding to the 95th percentile of cluster elements instead of its diameter, to deal with potential outliers. However, this is out of scope for this work.

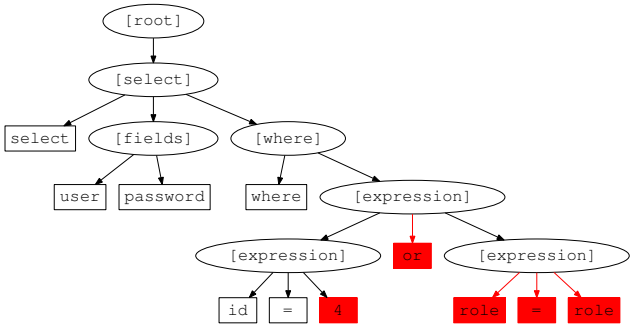
In our example, the diameter[4] of cluster 2 is equal to 0. We compare the distance between the test T and the nearest medoid (distance is 11) with the diameter (equals to 0). When the distance to the medoid is smaller than or equal to the diameter, the test is deemed to fit this cluster and it is classified as a safe execution. However, in our example the test falls outside of the cluster border (distance $>$ diameter) and is classified as a potential attack.

Cluster	Medoid	Diameter	Test distance
1	stmt1	0	18
2	stmt3	0	11

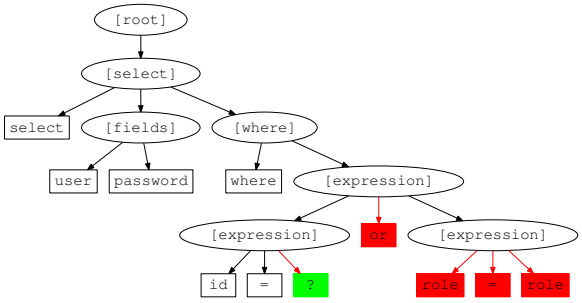
(a) Cluster diameters and distance

select user, password **from** users
where id = 4 **or** role = role;

(b) Statement



(c) Parse tree



(d) Pruned parse tree

Figure 8.6. Classification of a malicious statement.

8.4 Experimental Evaluation

This section presents the experimental evaluation designed and conducted to assess the proposed security oracle.

8.4.1 Research Questions and Variable Selection

The first research question concerns the accuracy of the oracle in classifying SQL statements. Missed attacks may lead to unaddressed security defects and false alarms lead to significant wasted effort, which should remain within reasonable bounds.

RQ$_1$: *How accurate is SOFIA in classifying legitimate SQL statements and SQLi attacks?*

The second research question is about the amount of time taken by the classifier to make a decision on a newly observed SQL statement. Fast run-time classification of attacks during testing is important to support efficient test automation.

RQ$_2$: *How fast is SOFIA in classifying SQL statements as legitimate or attacks?*

[4]Because of the small size of the running example, each cluster contains just one pruned parse tree, so diameter is equal to 0.

The third research questions is meant to compare SOFIA with available and comparable alternative solutions, which can be considered a baseline on which to improve, both in terms of classification accuracy and speed.

> **RQ$_3$**: *How does SOFIA compare to main-stream alternative tools in terms of accuracy and speed?*

Accuracy in our context is characterised by attack detection rate (we want to detect as many attacks as possible) and false positive rate. We need to minimise false positives as they trigger unnecessary manual analysis, which is expensive. The more false positives, the more analysis effort is being wasted and the scalability of the approach is being compromised. Accuracy is quantified by the standard *Recall* and *FPR* (False Positive Rate) metrics from information retrieval:

- *True Positives (TP):* The number of actual attacks that are correctly classified by the oracle as attacks;
- *False Positives (FP):* The number of legitimate statements that are incorrectly classified by the oracle as attacks;
- *True Negatives (TN):* The number of legitimate statements that are correctly classified by the oracle as safe;
- *False Negatives (FN):* The number of actual attacks that are incorrectly classified by the oracle as safe statements;
- **Recall:** The ratio between the correctly detected attacks and all the actual attacks:

$$\frac{TP}{TP+FN} \tag{8.2}$$

- **FPR:** The ratio between false positives and all the actual legitimate statements:

$$\frac{FP}{FP+TN} \tag{8.3}$$

Classification time is measured as follows:

- **C-Time:** Amount of time spent by the classifier to make a decision on a test outcome.

Time is measured by instrumenting the oracle. System time is probed before starting and after concluding the complete classification procedure for each SQL statement. It includes the amount of time spent for parsing the statement, pruning its tree, and the amount of time for computing its distances to the medoids of the safe model oracle.

The amount of time required to train the oracle is less interesting because training is done just once in a while and has therefore limited practical implications. It will not be further discussed here.

8.4.2 Subject Applications

The subject applications considered in this study are web applications and web services that use an SQL relational database. Moreover, since the security oracle will be assessed on real vulnerabilities, we selected applications by inspecting their bug-tracking systems and the *Common Vulnerabilities*

and Exposures repository[5] that keeps track of publicly known vulnerabilities and exposures. The chosen subject applications contain real SQL-injection vulnerabilities.

The applications considered in the study are:

- Hotel Reservation Service: written in PHP, Hotel Reservation Service is a service-oriented based system providing web services for room reservation. It was developed and used by Coffey et al. [Coffey et al., 2010a].
- SugarCRM: written in PHP, SugarCRM is a popular customer relationship management system[6].
- Taskfreak: written in PHP, Taskfreak is a web project management application[7].
- TheOrganizer: a web application that supports management and organisation of the activities in a personal agenda[8]. The server is written in Java (using Servlets, J2EE and Spring JDBC).
- Wordpress: written in PHP, Wordpress is a popular blogging and news publishing platform[9]. Wordpress has many utility plugins that are vulnerable to SQLi . We have two variants of Wordpress, one with the *newstatpress* plugin[10] that provides access statistics to Wordpress; the other with *landing-pages*[11], a plugin for customising templates and attracting more visits to blogging sites.

We use the test suites of the subject applications for generating training data. Wordpress comes with a large test suite of more than 3700 phpunit test cases. For Taskfreak and TheOrganizer, we reuse the test suites generated by available techniques [Tonella et al., 2014, Nguyen et al., 2012] and its accompanying tool[12]. For the remaining two, HRS and SugarCRM, we manually defined test suites that exercise all operations of their web services with various domain inputs.

Our reliance on real vulnerabilities in applications makes our results more representative of the current situation though it is impossible to predict what these results will be with future types of vulnerabilities. However, as described above, because our approach does not learn from specific attacks and relies on learning to characterise safe statements, we hope that the safe model will be able to handle future types of vulnerabilities as well.

8.4.3 Attack Generation

To evaluate the classifier, both legitimate executions and attacks are required. However, our oracle is independent of the input data generation strategy adopted to generate the attacks. Thus, we will evaluate the accuracy of the oracle with diverse attack generation tools:

- Burpsuite : A commercial security testing tool suite[13]. It has a vulnerability scanner that targets many types of vulnerabilities, including those in the OWASP top 10 [Williams and Wichers,

[5] https://cve.mitre.org
[6] http://www.sugarcrm.com
[7] http://www.taskfreak.com
[8] http://www.apress.com/9781590596951
[9] https://wordpress.org
[10] https://wordpress.org/plugins/newstatpress
[11] https://wordpress.org/plugins/landing-pages
[12] http://selab.fbk.eu/magic
[13] http://portswigger.net/burp

2013]. For detecting SQLi , Burpsuite (version 1.6.23) has a fixed list of 134 built-in SQLi test payloads, such as:

'a or 1=1−− | //∗ | replace | drop table

When scanning for SQLi vulnerabilities, Burpsuite uses these payloads as request parameters and submits them to a target system. It then analyses the obtained responses from the system to detect SQL code or error messages in order to report SQLi issues.

- SqlMap : A popular open source tool for penetration testers to detect and exploit SQLi vulnerabilities[14]. It supports various database management systems and implements many heuristics to generate test payloads for different types of SQLi, including *boolean-based blind, time-based blind, error-based, UNION query-based, stacked queries and out-of-band*.
- Xavier : A framework for the automated testing of web services for SQLi vulnerabilities [Appelt et al., 2014, Appelt et al., 2015]. Powered by a grammar developed specifically for SQLi attacks and machine learning, Xavier can generate diverse test payloads that can bypass web application firewalls and detect SQLi vulnerabilities.

8.4.4 Alternative Oracles

Apart from assessing SOFIA on diverse applications, it is interesting to investigate how it fares when compared to existing tools with similar goals. We found only two alternative tools: AntiSQL and GreenSQL [15].

AntiSQL is a tool provided by the vendor of the SQL parser we use in our work, which takes log files containing SQL statements as inputs, and reports whether their content is classified as attacks or legitimate statements.

GreenSQL is a popular database security solution for controlling database accesses, blocking SQLi attacks, among other features. It intercepts communications between applications and databases, learns patterns of regular SQL statements, and then, blocks malicious ones from getting to databases under protection.

8.4.5 Experimental Procedure

To collect SQL statements, we install and configure the subject applications, each on a separate virtual machine having *mysql* and *mysql-proxy* ready. *mysql-proxy* helps intercept and log all the SQL statements that an application sends to its database. We, then, execute the test suite that comes with each application to collect legitimate executions, i.e., safe statements. After that, we run the attack tools to generate attacks. The logs of attack tools are manually analysed and statements are labelled as attack or safe. Such analysis is required as safe statements can result from attacks since the system might perform routine updates or run additional queries before executing the attack statements. Note that the labelling task is only relevant for our assessment purposes; it is not needed in the real usage of the oracle.

[14] http://sqlmap.org

[15] http://www.greensql.com

We adopt a 10-fold validation strategy to check our oracle on different partitions of training and testing data. The training data of each subject application, consisting exclusively of safe statements, is divided randomly into 10 sets of approximately the same size, to form 10 partitions:

- Nine sets of legitimate statements represent the *training set*. They are used to train the safe model of the oracle. In our case, this is done only on legitimate statements and there is no need to split attacks across training and testing sets;
- The remaining set of legitimate statements is merged with the attacks to form the *testing set*. The oracle is used to classify each entry in the testing set using the safe model.

This process is iterated 10 times, once per each of the 10 possible partitions. The classification elaborated by the oracle for the testing set is compared with the actual labelling, to evaluate the oracle accuracy. The 10-fold validation, including training and testing for all subject applications, was executed using a HPC (high-performance computing) system, where the CPU speed on nodes is 2.26GHz, and 4Gb RAM was available to each process.

8.4.6 Experimental Results

Table 8.2 reports the number of SQL statements per application that have been considered in our study. The first two columns contain the name of the application and the tool used to generate the attacks, respectively. The subsequent columns report the number of legitimate statements and the number of successful attacks. The last two columns indicate the number of distinct pruned trees for legitimate statements and attacks. We cannot explore all combinations subject applications and tools because of certain application characteristics (web-based and web services) and the intended usage of the tools: Xavier targets web services while the others target standard web-based applications. In total, we obtain nine datasets for the experimental evaluation.

Table 8.2. Summary of the datasets used in our experiment: nine datasets obtained from six applications and three attack tools.

Application	T. Tool	#Legit.	#Attack	#Pruned Legit.	#Pruned Attack
HotelRS	Xavier	10,392	1,871	2,124	442
SugarCRM	Xavier	100,683	196	52	78
Taskfreak	burpsuite	7,502	3	29	2
Taskfreak	sqlmap	7,503	4	30	2
Theorganizer	burpsuite	1,516	28	27	17
Theorganizer	sqlmap	1,616	27	25	18
Wordpress-newstatpress	burpsuite	196,556	314	860	277
Wordpress-newstatpress	sqlmap	148,325	4	809	2
Wordpress-landingpage	sqlmap	171,487	170	843	65

We can observe, based on the data shown in Table 8.2, that the datasets used in our experiments are diverse in terms of the number of safe statements, ranging from 1k to 170k. Likewise, the number of attack statements generated by different tools varies from three to 1,871 attacks. Also, we can see that the number of parse trees after pruning is significantly reduced. For example, for subject *Wordpress-landingpage* and tool *sqlmap*, there were originally more than 171k safe statements and

170 attacks. However, after pruning, there are only 843 safe and 65 attack cases left, respectively. Overall, the percentage of reduction for all subject applications after tree pruning ranges from 79% to more than 99%. As a result, pruning helps reducing the time required by SOFIA, especially for training. Note that *burpsuite* cannot generate any attack on *Wordpress-landingpage* and, therefore, this pair is not investigated.

Table 8.3 provides experimental results. For each application, the performance of SOFIA is measured using Recall, false positive rate, and C-Time, as previously described. The values in the table represent the average over the 10 partitions of training/testing data and executions.

Table 8.3. Results of our approach: data averaged from 10-fold cross validation. *T.(ms)* is classification C-Time measured in millisecond.

App./*tool*	**TP**	**FP**	**TN**	**FN**	**Recall**	**FPR**	**T.(ms)**
HotelRS/ *Xavier*	1,871	0.0	1,039.2	0	**1.0**	**0.000**	25.14
SugarCRM/ *Xavier*	196	1.0	10,067.3	0	**1.0**	**0.000**	463.77
Taskfreak/ *burpsuite*	3	0.3	749.9	0	**1.0**	**0.000**	48.13
Taskfreak/ *sqlmap*	4	0.4	749.9	0	**1.0**	**0.001**	152.91
Theorganizer/ *burpsuite*	28	0.9	150.7	0	**1.0**	**0.006**	26.11
Theorganizer/ *sqlmap*	27	0.5	161.1	0	**1.0**	**0.003**	29.07
Wordpress-newstat/ *burpsuite*	4	28.5	14,804.0	0	**1.0**	**0.002**	33.70
Wordpress-newstat/ *sqlmap*	170	29.3	17,119.4	0	**1.0**	**0.002**	28.92
Wordpress-landingpage/ *sqlmap*	314	28.0	19,627.6	0	**1.0**	**0.001**	20.30

Regarding $\mathbf{RQ}_1$, results in Table 8.3 show that SOFIA yields a perfect Recall of 100% for all the subject applications and achieves a very low false positive rate across all applications (0.006 at the highest). When we consider the absolute number of false positives (FP), they are mostly below 1.0 on average. The highest FP is only 29.3, therefore suggesting that manual analysis is feasible even in the worst case.

Thus we can provide a clear answer to $\mathbf{RQ}_1$:

SOFIA delivers a near-perfect accuracy in classifying both legitimate SQL statements and attacks.

Moreover, considering $\mathbf{RQ}_2$, the time required to classify a new statement is small, with an average across case studies of 92ms and a median of 29ms. For most of the case studies, classification takes around 30ms per statement, with the exception of Sugar-xavier that takes more than 400ms. The reason for this difference is that, on average, the parse trees of SQL statements used by Sugar are larger and thus lead to longer tree-edit distance calculations.

Thus we can answer $\mathbf{RQ}_2$:

SOFIA is fast in classifying SQL statements, taking on average 92ms per classification.

Furthermore, to answer $\mathbf{RQ}_3$, we compared SOFIA to AntiSQL and GreenSQL .

Following the same procedure as for SOFIA, we ran these two industrial tools against all nine datasets and measured their accuracy and time. GreenSQL was given the same training data that had been used to train SOFIA. AntiSQL inspects SQL statements based on its own SQLi filters and, therefore, no training was needed. The same testing data were then checked by GreenSQL and AntiSQL . For each dataset, TP and FN were measured by counting the number of attack SQL statements correctly classified as attacks or incorrectly classified as safe, respectively. Likewise, we measured the average number of safe statements classified as attacks (FP) and safe (TN). Further, we measured the average execution time the tools required to parse and check a statement.

Table 8.4. Results of AntiSQL and GreenSQL . *T.(ms)* is the average time in millisecond the tools need to process one statement.

App/*tool*	TP	FP	TN	FN	Recall	FPR	T.(ms)
AntiSQL							
HotelRS/ *Xavier*	1,579	827.0	212.2	292	**0.84**	**0.796**	320
SugarCRM/ *Xavier*	50	1,237.6	8,830.7	146	**0.25**	**0.123**	314
Taskfreak/ *burpsuite*	1	12.9	737.3	2	**0.33**	**0.017**	250
Taskfreak/ *sqlmap*	4	13.0	737.3	0	**1.00**	**0.017**	303
Theorganizer/ *burpsuite*	28	123.0	28.6	0	**1.00**	**0.811**	302
Theorganizer/ *sqlmap*	27	133.0	28.6	0	**1.00**	**0.823**	312
Wordpress-newstatpress/ *burpsuite*	0	5,389.6	9,442.9	4	**0**	**0.363**	306
Wordpress-newstatpress/ *sqlmap*	154	5.562.0	11,586.7	16	**0.91**	**0.324**	294
Wordpress-landingpage/ *sqlmap*	155	5,682.0	13,973.6	159	**0.49**	**0.289**	300
GreenSQL							
HotelRS/ *Xavier*	1,871	0.0	1.039.2	0	**1.0**	**0.000**	6
SugarCRM/ *Xavier*	133	2,020.6	8.047.7	63	**0.68**	**0.201**	5
Taskfreak/ *burpsuite*	3	0.3	749.9	0	**1.0**	**0.000**	6
Taskfreak/ *sqlmap*	4	0.4	749.9	0	**1.0**	**0.001**	5
Theorganizer/ *burpsuite*	28	59.1	92.5	0	**1.0**	**0.390**	5
Theorganizer/ *sqlmap*	27	58.7	102.9	0	**1.0**	**0.363**	5
Wordpress-newstatpress/ *burpsuite*	4	1,372.0	13,460.0	0	**1.0**	**0.093**	5
Wordpress-newstatpress/ *sqlmap*	170	1,360.4	15,788.3	0	**1.0**	**0.079**	5
Wordpress-landingpage/ *sqlmap*	314	1,671.6	17,984.0	0	**1.0**	**0.085**	5

Table 8.4 shows the Recall, FPR, and execution time of AntiSQL and GreenSQL . We can observe that FPR and Recall of AntiSQL are significantly worse compared to those of SOFIA. AntiSQL missed many attacks on all the case studies (FN > 0) and, as a result, Recall of some cases is very low. It wrongly classified safe statements as attacks (high FP rate) on many cases, leading to a poor FPR of 0.823 (or 82.3%). In particular, no attack was correctly identified on one case (TP = 0). GreenSQL is as good as SOFIA in all but one subject (SugarCRM) with respect to Recall. However, GreenSQL reported many false positives that resulted in an order of magnitude higher FPR (up to 0.363) as compared to that of SOFIA (0.006).

Regarding the time taken to process an SQL file, AntiSQL takes on average 300ms, which is higher than the average time required by SOFIA (92ms). GreenSQL takes only about 5ms and is therefore faster than SOFIA at the cost of a much higher number of false positives. It is worth noticing that GreenSQL is a leading industrial tool while SOFIA is a currently research prototype. Besides, because of technical reasons, GreenSQL could not run on the HPC but on a server that happened to

have a higher CPU frequency than the computer used for SOFIA and AntiSQL . This setting clearly favoured GreenSQL in detecting attacks faster and prevents us from drawing objective conclusions regarding its comparison with SOFIA regarding its run-time speed.

To compare accuracy and classification time, we used the non-parametric Wilcoxon test. The use of non-parametric tests requires no distributional assumption. Such a test checks whether differences in performance recorded for SOFIA and AntiSQL are statistically significant[16]. Results show that SOFIA performs significantly better than AntiSQL with respect to Recall, FPR and time (*p-values* are, respectively, 0.028, 0.008 and 0.011). Similarly, SOFIA fares significantly better than GreenSQL in terms of FPR (*p-value* = 0.028). Thus, we can provide a clear answer to $\mathbf{RQ}_3$:

SOFIA is significantly more accurate than AntiSQL and GreenSQL and significantly faster than AntiSQL in classifying legitimate SQL statements and SQLi attacks.

8.4.7 Threats to Validity

To help increase the external validity of our results, which is the main challenge in our study, we relied on various applications from different domains and written using different programming languages, and three different attack generation tools. Further, to ensure our accuracy results were realistic, we resorted to standard 10-fold validation involving multiple training/testing data sets. However, we have to recognise the inherent limitations of such studies, as we cannot predict accuracy on future vulnerabilities. The fact that our learning approach does not rely on the specific vulnerabilities in our application systems helps alleviate this problem but does not eliminate it entirely.

8.5 Related Work

We review SQLi detection techniques that are based on analysing SQL statements at run-time. Table 8.5 summarizes which of our security oracle requirements (see Section 8.2.1) are met by related work. Our requirements are: Req_1 training is independent from known attacks; Req_2 classification is neither based on the knowledge of test input data nor on the input data generation algorithm; and Req_3 analysis does not require access to source code.

Table 8.5. Security oracle requirements met by related work.

Papers	Req $_1$	Req $_2$	Req$_3$
Halfond et al. [Halfond and Orso, 2005a] Bisht et al. [Bisht et al., 2010] Buehrer et al. [Buehrer et al., 2005] Kemalis et al. [Kemalis and Tzouramanis, 2008]	✓	✓	–
Pinzon et al. [Pinzón et al., 2013]	–	✓	✓
Liu et al. [Liu et al., 2009] Valeur et al. [Valeur et al., 2005]	✓	✓	✓

Whitebox approaches [Halfond and Orso, 2005a, Bisht et al., 2010, Buehrer et al., 2005, Kemalis and Tzouramanis, 2008] require the source code to be available for instrumentation or analysis, so

[16] We assume a 95% confidence level, so a p-value < 0.05 indicates a statistically significant result.

they do not meet requirement Req_3. Halfond et al. proposed AMNESIA [Halfond and Orso, 2005a], a method based on program analysis to build models (non-deterministic finite automata) for each and every legitimate query of an application. The application is instrumented and each SQL query sent to the database is validated by finding an accepting path in the automaton. If not possible, the query is considered to be an attack. Bisht et al. proposed CANDID [Bisht et al., 2010], an approach that compares a developer's intended query structure with the actual query structure found during program execution. While both AMNESIA and CANDID show promising evaluation results and they meet requirement Req_1 and Req_2, but they require access to the source code and its instrumentation, which limits their applicability and violates requirement Req_3.

Buehrer et al. have proposed SQLGuard, which compares parse trees of each SQL statement before and after the inclusion of user inputs at runtime [Buehrer et al., 2005]. If the trees corresponding to a statement (with and without user inputs) are different after removing constants, then the statement is considered to result from SQLi attacks. In comparison to our approach, which does not require any change to the source code, the application of SQLGuard requires SQL queries to be rewritten using a Java library provided by SQLGuard's authors, thus Req_3 is not fully met.

Several approaches based on anomaly detection have been proposed in the literature [Kemalis and Tzouramanis, 2008, Liu et al., 2009, Pinzón et al., 2013, Valeur et al., 2005]. Many of them look similar to ours because they contain the same two high-level steps: training and detection. We provide below a detailed comparison with our work, which aims at addressing three main limitations of practical importance: false positives, the difficulty to obtain a somewhat complete set of of actual and varied attacks for learning purposes, and the need to handle new attack variants.

Kemalis et al. proposed SQL-IDS, a specification-based approach to detect malicious SQL statements [Kemalis and Tzouramanis, 2008]. Even if this approach does analyse source code directly, the user has to provide the specification of all benign SQL statements for the application under protection. SQL-IDS monitors the application during runtime and each query that does not comply with the specification is treated as malicious. Req_3 is not fully met by this approach, because it requires precise knowledge of the source code to be manually provided to the tool. In our proposed approach, the user does not have to provide any specification for benign statements, because the safe model is automatically inferred from the learning set.

The remaining approaches are black box, i.e. they do not require source code, so they meet requirement Req_3. Pinzon et al. proposed an anomaly detection approach combining neural networks and support vector machine to classify SQL queries into benign or malicious statements [Pinzón et al., 2013]. In contrast to our approach, they employ supervised machine learning techniques that require a sufficient number of known attack statements. As it is very much driven by what known attacks were fed to the learner, such an approach does not met requirement Req_1 and it might have difficulties recognising new attacks.

To the best of our knowledge, only two approaches [Liu et al., 2009] [Valeur et al., 2005] fully meet all of our three requirements. However, we overcome their limitations by achieving (i) low sensitivity to learning set incompleteness and (ii) a very low false positive rate.

Liu et al. proposed SQLProb [Liu et al., 2009], a tool that uses string alignment to detect the part of an SQL query that corresponds to user input, by detecting the difference between a new query

(requirements Req_2 and Req_3) and all the queries observed at learning time (requirement Req_1). The tool reports an anomaly when the part of the parse tree that corresponds to detected user input contains non-constant leaf nodes (e.g., arithmetic or logic operators). As one might expect, the reliability of this approach is very sensitive to the completeness of learning, which is whether all types of queries are accounted for. Completeness is required to identify correctly the user input in the SQL query. Identifying user inputs is a difficult and error-prone step that could lead to false positives when training is partial. Unfortunately, the authors did not report false positives. After investigation, it turned out that the tool was not available and therefore a comparative study was not possible. Nevertheless, our approach does not entail extracting user inputs and is therefore by design more robust. We do not need all types and variants of safe queries to be available at the training phase and, thanks to the distance measure that we adopt, as demonstrated by our empirical study, we obtain accurate results even when we have no guarantee of completeness during training[17].

Valeur et al. [Valeur et al., 2005] used machine learning to learn relevant characteristics from user inputs of benign SQL queries. In the training phase, several statistical models characterising relevant features, such as character distribution and string length, are learned from attack-free SQL queries (requirement Req_1), in order to capture patterns and ranges of expected values. In the detection phase, a query is intercepted (requirements Req_2 and Req_3) and parsed, and its input values are compared to detect anomalies against models resulting from training queries with identical parse tree structure. This approach requires an exact match between the parse tree to classify and those in the learning set, while we tolerate a degree of difference using tree distance. Moreover, Valeur et al. learn statistical distributions of values from legitimate SQL statements, while we remove these values and consider only pruned parse trees. These two fundamental differences allow us to dramatically reduce false positives based on results reported by Valeur et al. In our approach, legitimate statements are correctly classified as safe if they belong (i.e., show acceptable distance) to one of the clusters of our model. Our approach is therefore more resilient, as demonstrated by our empirical results.

8.6 Summary

Having in mind realistic industrial settings, we elicited three requirements for an ideal security oracle for security testing of SQLi vulnerabilities. We presented a novel approach that satisfies all of these requirements and we implemented it into a tool that we call SOFIA.

SOFIA learns a safe model characterising legitimate SQL statements, based on information logged during normal system executions. To do so, SQL statements are parsed and then parse trees are pruned to remove information that is irrelevant to whether an SQL statement is safe or not. Based on their tree-edit distance, similar parse trees are grouped using clustering and the resulting set of clusters is used as a safe models. SOFIA classifies new SQL statements by comparing and contrasting them with these clusters. Executions whose SQL statements do not sufficiently fit into any of the clusters of the safe model are classified as attacks.

We assessed the accuracy of SOFIA as a security oracle with three different attack generation tools on six PHP and Java systems. No attack was missed and the rate of false positives was very

[17] Recall that our training set of queries is based on functional test suites, when available, and otherwise monitoring usage.

low, thus making SOFIA a reliable and cost-effective approach. Further, the classification of SQL statements was on average below 100ms, thus making it possible to execute a large number of test cases within time constraints. Last, SOFIA significantly outperformed two widely used alternative tools in terms of classification accuracy and execution speed for one of the tools.

Chapter 9

Conclusions and Future Work

This chapter summarises our research contributions and gives a perspective for future research activities in the area.

9.1 Summary

This dissertation presents several security testing approaches to assess web applications and services for SQLi vulnerabilities. Our work was done in collaboration with SIX Payment Services, a leading financial service provider in Luxembourg and Switzerland. We analysed the challenges of security testing in an industrial context and we identified several factors that might limit the applicability or effectiveness of existing security testing approaches in such a context. The key limiting factors are:

- The source code of the core business application used at SIX can typically not be analysed or modified for the purpose of testing, because several components are developed by external parties. However, the interactions between the application and its environment (e.g. network or database calls) can be monitored to guide the testing. For this purpose, common components in corporate IT environments, e.g. DIDS, can be leveraged to increase the accuracy of test oracles.
- Industry standards such as PCI-DSS require extensive testing efforts. Given time and resource constraints, manual labour becomes a bottleneck and security properties are often not sufficiently tested. Testing techniques with a high degree of automation can mitigate the lack of manual labour.
- Payment card processors, such as our industrial partner, typically use WAFs to protect their web services from cyberattacks. Testing techniques should consider WAFs when assessing such web services for vulnerabilities, since vulnerabilities that are exploitable despite a WAF being in place are more severe and, thus, should be prioritised. Furthermore, testing techniques should systematically check the attack detection capabilities of WAFs in order to provide effective protection and prevent web service from being exploited.

To address these issues, this dissertation proposes several security testing approaches that are **automated**, applicable in **black box** testing scenarios, able to **assess and bypass WAFs**, and use an **accurate test oracle**, which is based on DIDSs. We empirically evaluated each proposed approach by conducting an industrial case study and/or by using popular open source software as subject applications.

Chapter 4 assessed the influence of common components in corporate IT environments, namely WAFs and DIDSs, on the effectiveness of black box security testing approaches. We proposed that testing through a WAF can be used to detect vulnerabilities in web services that are insufficiently protected and, thus, should be prioritised in debugging efforts. Furthermore, we proposed to base test oracles on DIDS to achieve a higher effectiveness and efficiency. In the evaluation, we compared our prototype implementation to SqlMap, a state-of-the-art penetration testing tool. We found that testing through a WAF can indeed be used to prioritise vulnerabilities and a DIDS can increase the effectiveness and efficiency of black box testing techniques. Our prototype detected more vulnerabilities than SqlMap, specifically if the subject application was protected by a WAF, and it required fewer tries until a vulnerability was detected. However, we also concluded that if a WAF protected the web services our prototype could not generate SQLi attacks that lead to syntactically correct SQL statements and, hence, it was unclear if the vulnerabilities were exploitable.

Chapter 5 built on the idea of using a DIDS as test oracle and combined it with an input generation approach that can generate a diverse set of SQLi attacks. Starting from "legal" initial test cases, the input generation approach applies a set of mutation operators that are specifically designed to increase the likelihood of generating syntactically correct SQL statements that can reveal SQLi vulnerabilities. In our evaluation, we compared our input generation approach to a baseline approach and found it is faster and significantly more likely to detect vulnerabilities within a limited time budget. Moreover, when the subject systems are protected by a WAF, our approach can still generate a good amount of attacks that get through the firewall and lead to executable SQL statements for all-but-one vulnerabilities.

In Chapter 6, we focused our research on testing WAFs. We presented a machine learning-driven testing approach to automatically detect holes in WAFs that let SQL injection attacks bypass them. At the beginning, the approach automatically generates a diverse set of attacks and then submits them to a system that is protected by a WAF. Incrementally learning from the attacks that are blocked or bypassing the WAF, our approach selects attacks that exhibit characteristics associated with bypassing the WAF and mutates them to efficiently generate new bypassing attacks. We developed a tool that implements the approach and evaluated it on ModSecurity, a widely used WAF, and a proprietary WAF of our industrial partner. Our evaluation indicated that our proposed technique is efficient at generating SQL injection attacks that can bypass a WAF and can be used to identify successful attack patterns.

Chapter 7 introduced an approach to infer a WAF filter rule from the bypassing attacks that were identified with our proposed WAF testing approach presented in the previous chapter. The goal of the inferred filter rule is to identify all bypassing attacks without matching any legitimate request. When such a filter rule is added to the WAF's rule set, the previously bypassing attacks are correctly identified without adversely affecting the legitimate usage of the system. Experimental results showed that the generated filter rule is effective at blocking the bypassing attacks (*recall* is between 49% and 96%), while inducing a small number of false positives (*false positive rate* is between 0% and 3%) and, thus, is effective at repairing a WAF.

Chapter 8 presented a test oracle that was specifically designed to meet the requirements of testing for SQLi vulnerabilities in an industrial context. The oracle is programming-language and source-code independent, and can be used with various attack generation tools. Moreover, because it does

not rely on known attacks, the oracle is meant to also detect types of SQLi attacks that might be unknown at learning time. We proposed to recast the oracle challenge as a one-class classification problem where we learn to characterise legitimate SQL statements to accurately distinguish them from SQLi attack statements. We have carried out an experimental validation on six applications, of which two are large and widely used. The oracle was used to detect real SQLi vulnerabilities with inputs generated by three attack generation tools. The obtained results show that the oracle is computationally fast and achieves a recall rate of 100% (i.e., missing no attacks) with a low false positive rate (0.6%).

9.2 Future Work

In today's applications, SQL injection vulnerabilities are common and testing for SQLi is of high importance. However, future application frameworks might embrace alternative technologies and other injection vulnerabilities specific to these technologies might become more important.

Injection vulnerabilities follow a common pattern: An injection vulnerability occurs if an application uses untrusted input data that was not sanitised to build commands for an external interpreter, e.g. Unix Shells, script interpreters such as Perl or Python, or LDAP. This dissertation presents testing approaches specific to SQL injection, but the basic principals and techniques can be applied to other kinds of injection vulnerabilities. For example, we increased the accuracy of test oracles for SQLi by monitoring the SQL statements an application under test sends to the database. By doing so, our oracle can precisely identify if a SQLi test case resulted in a malicious SQL statement and, thus, decide if the application is vulnerable. Similarly, this concept can be applied to other injection vulnerabilities like XPATH or LDAP injection. Commands sent to the corresponding interpreters could be monitored (e.g. by using network proxies, system call interposing, or setting the application under test to a verbose log level) and malicious queries could be detected in a similar fashion as in our proposed SQLi test oracle. Further research is necessary to devise input generation techniques; for example, a set of input mutation operators such as $\mu 4SQL$, for other injection vulnerabilities. In addition, further research is necessary to evaluate if our clustering methodology proposed in SOFIA is suitable in the context of other injection vulnerabilities.

Our proposed technique to test WAFs can also be extended to assess the detection rate of other kinds of attacks. The SQLi attacks of our approach are generated from a context-free grammar. However, the approach is not limited to a specific grammar, but can work with any context-free grammar. Therefore, similar grammars can be formulated for other kinds of attacks and simply be replaced with the SQLi grammar. The implementation of our technique allows for the easy adaptation of other attack grammars, since the grammar is defined in an external text file that can be swapped. Having several attack grammars available would increases the test coverage of a WAF's rule set and an IT Security Engineer could choose the attack grammars that fits best the risk profile of the protected system. Further research is necessary to devise attack grammars for other vulnerabilities that accurately reflect attacks on applications in practice and account for the various obfuscation techniques used by attackers.

Bibliography

[bru, 2010] (2010). Verified firewall policy transformations for test case generation. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 345–354. IEEE.

[ora, 2014] (2014).

[Aggarwal and Reddy, 2013] Aggarwal, C. C. and Reddy, C. K. (2013). *Data clustering: algorithms and applications*. CRC Press.

[Al-Shaer et al., 2009] Al-Shaer, E., El-Atawy, A., and Samak, T. (2009). Automated pseudo-live testing of firewall configuration enforcement. *Selected Areas in Communications, IEEE Journal on*, 27(3):302–314.

[Anand et al., 2013] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., and McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001.

[Antunes et al.,] Antunes, N., Laranjeiro, N., Vieira, M., and Madeira, H. Command injection vulnerability scanner for web services. http://eden.dei.uc.pt/ mvieira/.

[Antunes et al., 2009] Antunes, N., Laranjeiro, N., Vieira, M., and Madeira, H. (2009). Effective detection of SQL/XPath injection vulnerabilities in web services. In *Proceedings of the 6th IEEE International Conference on Services Computing (SCC '09)*, pages 260–267.

[Antunes and Vieira, 2009] Antunes, N. and Vieira, M. (2009). Detecting SQL injection vulnerabilities in web services. In *Proceedings of the 4th Latin-American Symposium on Dependable Computing (LADC '09)*, pages 17–24.

[Appelt et al., 2013] Appelt, D., Alshahwan, N., and Briand, L. (2013). Assessing the impact of firewalls and database proxies on sql injection testing. In *Proceedings of the 1st International Workshop on Future Internet Testing*.

[Appelt et al., 2015] Appelt, D., Nguyen, C., and Briand, L. (2015). Behind an application firewall, are we safe from sql injection attacks? In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10.

[Appelt et al., 2014] Appelt, D., Nguyen, C. D., Briand, L. C., and Alshahwan, N. (2014). Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA. ACM.

[Avancini and Ceccato, 2013] Avancini, A. and Ceccato, M. (2013). Security oracle based on tree kernel methods. In *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, pages 30–43. Springer.

[Banzhaf et al., 1998] Banzhaf, W., Francone, F. D., Keller, R. E., and Nordin, P. (1998). *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Barr et al., 2015] Barr, E., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525.

[Bartolini et al., 2009] Bartolini, C., Bertolino, A., Marchetti, E., and Polini, A. (2009). Ws-taxi: A wsdl-based testing tool for web services. In *ICST*, pages 326–335.

[Bau et al., 2010] Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*, pages 332–345.

[Beery and Niv, 2013] Beery, T. and Niv, N. (2013). Web application attack report.

[Bertino et al., 2005] Bertino, E., Terzi, E., Kamra, A., and Vakali, A. (2005). Intrusion detection in rbac-administered databases. In *Computer security applications conference, 21st annual*, pages 10–pp. IEEE.

[Bisht et al., 2010] Bisht, P., Madhusudan, P., and Venkatakrishnan, V. (2010). Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):14.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[Buehrer et al., 2005] Buehrer, G., Weide, B. W., and Sivilotti, P. A. (2005). Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM.

[Chen, 2015] Chen, S. (2015). Price and feature comparison of web application scanners.

[Christey and Martin, 2007] Christey, S. and Martin, R. A. (2007). Vulnerability type distributions in cve. http://cwe.mitre.org.

[Chung et al., 2000] Chung, C. Y., Gertz, M., and Levitt, K. (2000). Demids: A misuse detection system for database systems. In *Integrity and Internal Control in Information Systems*, pages 159–178. Springer.

[Ciampa et al., 2010] Ciampa, A., Visaggio, C. A., and Di Penta, M. (2010). A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems (SESS '10)*, pages 43–49.

[Coffey et al., 2010a] Coffey, J., White, L., Wilde, N., and Simmons, S. (2010a). Locating software features in a soa composite application. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 99–106.

[Coffey et al., 2010b] Coffey, J., White, L., Wilde, N., and Simmons, S. (2010b). Locating software features in a soa composite application. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS '10)*, pages 99–106.

[Damele et al., 2013] Damele, B., Guimaraes, A., and Stampar, M. (2013). Sqlmap. http://sqlmap.org/.

[Deb et al., 2002] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197.

[Desmet et al., 2006] Desmet, L., Piessens, F., Joosen, W., and Verbaeten, P. (2006). Bridging the gap between web application firewalls and web applications. In *Proceedings of the fourth ACM workshop on Formal methods in security*, pages 67–77. ACM.

[Doupé et al., 2010] Doupé, A., Cova, M., and Vigna, G. (2010). Why johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '10)*, pages 111–131.

[Efron and Tibshirani, 1993] Efron, B. and Tibshirani, R. (1993). *An Introduction To The Bootstrap*, volume 57. CRC press.

[Elia et al., 2010] Elia, I. A., Fonseca, J., and Vieira, M. (2010). Comparing sql injection detection tools using attack injection: An experimental study. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, pages 289–298.

[Felderer et al., 2015] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2015). Security testing: A survey. *Advances in Computers*.

[Fonseca et al., 2007] Fonseca, J., Vieira, M., and Madeira, H. (2007). Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC '07)*, pages 365–372.

[Fonseca et al., 2014] Fonseca, J., Vieira, M., and Madeira, H. (2014). Evaluation of web security mechanisms using vulnerability & attack injection. *Dependable and Secure Computing, IEEE Transactions on*, 11(5):440–453.

[Forristal, 1998] Forristal, J. (1998). Nt web technology vulnerabilities.

[Fossi and Johnson, 2009] Fossi, M. and Johnson, E. (2009). Symantec global internet security threat report, volume xiv.

[Fu and Qian, 2008] Fu, X. and Qian, K. (2008). SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB '08)*, pages 34–39.

[GreenSQL LTD., 2013] GreenSQL LTD. (2013). Greensql. http://www.greensql.com.

[Halfond et al., 2006a] Halfond, W., Viegas, J., and Orso, A. (2006a). A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE.

[Halfond et al., 2009] Halfond, W. G., Anand, S., and Orso, A. (2009). Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 285–296.

[Halfond and Orso, 2005a] Halfond, W. G. and Orso, A. (2005a). Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM.

[Halfond et al., 2006b] Halfond, W. G., Viegas, J., and Orso, A. (2006b). A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE '06)*, pages 13–15.

[Halfond and Orso, 2005b] Halfond, W. G. J. and Orso, A. (2005b). Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pages 174–183.

[Halfond and Orso, 2006] Halfond, W. G. J. and Orso, A. (2006). Preventing SQL injection attacks using AMNESIA. In *Proceedings of the 28th International Conference on Software Engineering (ICSE' 06)*, pages 795–798.

[Hashemi et al., 2008] Hashemi, S., Yang, Y., Zabihzadeh, D., and Kangavari, M. (2008). Detecting intrusion transactions in databases using data item dependencies and anomaly analysis. *Expert Systems*, 25(5):460–473.

[Holler et al., 2012] Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *Proceedings of the 21st Usenix Security Symposium*.

[Huang et al., 2003] Huang, Y.-W., Huang, S.-K., Lin, T.-P., and Tsai, C.-H. (2003). Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*, pages 148–159.

[Hwang et al., 2008] Hwang, J., Xie, T., Chen, F., and Liu, A. X. (2008). Systematic structural testing of firewall policies. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 105–114. IEEE.

[Jain, 2010] Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666.

[Jia and Harman, 2011] Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

[Jürjens and Wimmel, 2001] Jürjens, J. and Wimmel, G. (2001). Specification-based testing of firewalls. In Bjørner, D., Broy, M., and Zamulin, A., editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 308–316. Springer Berlin Heidelberg.

[Kemalis and Tzouramanis, 2008] Kemalis, K. and Tzouramanis, T. (2008). Sql-ids: a specification-based approach for sql-injection detection. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 2153–2158. ACM.

[Khoury et al., 2011] Khoury, N., Zavarsky, P., Lindskog, D., and Ruhl, R. (2011). Testing and assessing web vulnerability scanners for persistent sql injection attacks. In *Proceedings of the 1st International Workshop on Security and Privacy Preserving in e-Societies (SeceS '11)*, pages 12–18.

[Kieyzun et al., 2009a] Kieyzun, A., Guo, P., Jayaraman, K., and Ernst, M. (2009a). Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 199 –209.

[Kieyzun et al., 2009b] Kieyzun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. (2009b). Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 199–209.

[Kindy and Pathan, 2011] Kindy, D. A. and Pathan, A.-S. K. (2011). A survey on sql injection: Vulnerabilities, attacks, and prevention techniques.

[Lee et al., 2012] Lee, I., Jeong, S., Yeo, S., and Moon, J. (2012). A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, 55(1):58–68.

[Liu et al., 2009] Liu, A., Yuan, Y., Wijesekera, D., and Stavrou, A. (2009). Sqlprob: A proxy-based architecture towards preventing sql injection attacks. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 2054–2061, New York, NY, USA. ACM.

[Liu and Kuan Tan, 2008] Liu, H. and Kuan Tan, H. B. (2008). Testing input validation in web applications through automated model recovery. *Journal of Systems and Software*, 81(2):222–233.

[Manning et al., 2008] Manning, C. D., Raghavan, P., Schütze, H., et al. (2008). *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge.

[McMinn, 2004] McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156.

[Mitchell, 1998] Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

[Nebro et al., 2015] Nebro, A. J., Durillo, J. J., and Vergne, M. (2015). Redesigning the jmetal multi-objective optimization framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, pages 1093–1100, New York, NY, USA. ACM.

[Nguyen et al., 2012] Nguyen, C. D., Marchetto, A., and Tonella, P. (2012). Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA. ACM.

[Offutt et al., 2004] Offutt, J., Wu, Y., Du, X., and Huang, H. (2004). Bypass testing of web applications. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 187–197. IEEE.

[PCI Security Standards Council, 2013] PCI Security Standards Council (2013). Pci data security standard (pci dss). https://www.pcisecuritystandards.org.

[Pierre Bourque, 2014] Pierre Bourque, R. E. F., editor (2014). *Guide to the Software Engineering Body of Knowledge Version 3 (SWEBOK)*. IEEE.

[Pinzón et al., 2013] Pinzón, C. I., De Paz, J. F., Herrero, Á., Corchado, E., Bajo, J., and Corchado, J. M. (2013). idmas-sql: intrusion detection based on mas to detect and block sql injection through data mining. *Information Sciences*, 231:15–31.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*, volume 1. Morgan kaufmann.

[Reynolds et al., 2006] Reynolds, A., Richards, G., de la Iglesia, B., and Rayward-Smith, V. (2006). Clustering rules: A comparison of partitioning and hierarchical clustering algorithms. *Journal of Mathematical Modelling and Algorithms*, 5(4):475–504.

[Santos et al., 2014] Santos, R. J., Bernardino, J., and Vieira, M. (2014). Approaches and challenges in database intrusion detection. *ACM SIGMOD Record*, 43(3):36–47.

[Sekar, 2009] Sekar, R. (2009). An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

[Senn et al., 2005] Senn, D., Basin, D., and Caronni, G. (2005). Firewall conformance testing. In Khendek, F. and Dssouli, R., editors, *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin Heidelberg.

[Shahriar and Zulkernine, 2008] Shahriar, H. and Zulkernine, M. (2008). MUSIC: Mutation-based SQL injection vulnerability checking. In *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, pages 77–86. IEEE.

[Shar et al., 2013] Shar, L. K., Tan, H. B. K., and Briand, L. (2013). Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 642–651.

[Shasha and Zhang, 1990] Shasha, D. and Zhang, K. (1990). Fast algorithms for the unit cost editing distance between trees. *Journal of algorithms*, 11(4):581–621.

[Shin, 2006] Shin, Y. (2006). Improving the identification of actual input manipulation vulnerabilities. In *Proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*.

[Shin et al., 2006] Shin, Y., Williams, L., and Xie, T. (2006). Sqlunitgen: Test case generation for sql injection detection. *North Carolina State University, Raleigh Technical report, NCSU CSC TR*, 21.

[Smith et al., 2010] Smith, B., Williams, L., and Austin, A. (2010). Idea: using system level testing for revealing SQL injection-related error message information leaks. In *Proceedings of the 2nd International Conference on Engineering Secure Software and Systems (ESSoS '10)*, pages 192–200.

[SQL Injection Wiki, 2013] SQL Injection Wiki (2013). SQL injection cheat sheet. http://www.sqlinjectionwiki.com/.

[Srivastava et al., 2006] Srivastava, A., Sural, S., and Majumdar, A. K. (2006). Weighted intra-transactional rule mining for database intrusion detection. In *Advances in Knowledge Discovery and Data Mining*, pages 611–620. Springer.

[The Open Web Application Security Project (OWASP), 2013] The Open Web Application Security Project (OWASP) (2013). Testing for SQL injection (owasp-dv-005). http://www.owasp.org.

[Tonella et al., 2014] Tonella, P., Tiella, R., and Nguyen, C. D. (2014). Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 562–572, New York, NY, USA. ACM.

[Valeur et al., 2005] Valeur, F., Mutz, D., and Vigna, G. (2005). A learning-based approach to the detection of sql attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140. Springer.

[Varrette et al., 2014] Varrette, S., Bouvry, P., Cartiaux, H., and Georgatos, F. (2014). Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy. IEEE.

[Vieira et al., 2009] Vieira, M., Antunes, N., and Madeira, H. (2009). Using web security scanners to detect vulnerabilities in web services. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks (DSN'09)*, pages 566–571.

[W3C, 2012] W3C (2012). Character entity references in HTML 4. http://www.w3.org/TR/html4/sgml/entities.html.

[Wassermann and Su, 2007] Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 32–41.

[Wei et al., 2006] Wei, K., Muthuprasanna, M., and Kothari, S. (2006). Preventing SQL injection attacks in stored procedures. In *Proceedings of the Australian Software Engineering Conference (ASWEC '06)*, pages 191–198.

[Williams and Wichers, 2013] Williams, J. and Wichers, D. (2013). Owasp, top 10, the ten most critical web application security risks. Technical report, The Open Web Application Security Project.

[Witten and Frank, 2011] Witten, I. H. and Frank, E. (2011). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[Wohlin et al., 2000] Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., and Wesslen, A. (2000). *The Experimentation in Software Engineering – An Introduction*. Kluwer.

[Wool, 2004] Wool, A. (2004). A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67.

[Wool, 2010] Wool, A. (2010). Trends in firewall configuration errors: Measuring the holes in swiss cheese. *Internet Computing, IEEE*, 14(4):58–65.

[Zitzler and Thiele, 1999] Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *evolutionary computation, IEEE transactions on*, 3(4):257–271.

Appendix A

Investigating the difference between ML-Driven B and ML-Driven D

As discussed in Section 6.1.3, the variable MAX_M controls how ML-Driven spends the test budget. For ML-Driven D, MAX_M is smaller and, thus, fewer tests are selected for mutation, but more mutants are generated per test. For ML-Driven B, MAX_M is larger and, thus, more tests are selected for mutation, but fewer mutants are generated per test. In total, both techniques spend the same test budget.

In earlier work [Appelt et al., 2015], we compared the two variants of ML-Driven: ML-Driven B ($MAX_M = 10$) and ML-Driven D ($MAX_M = 100$). This appendix summarises the comparison outcomes and investigates the reason behind the difference between ML-Driven B and ML-Driven D.

A.1 Performance Comparison

We evaluated ML-Driven with a popular open-source WAF, ModSecurity, by measuring how many bypassing attacks ML-Driven B and ML-Driven D can generate over time [Appelt et al., 2015]. To account for the degree of randomness involved in ML-Driven, we repeated each test run 10 times. Each time a new test was generated, we have noted the passing wall-clock time and then executed it to see whether or not it could bypass the WAF. We compare the performance of the techniques based on the cumulative number of bypassing tests generated over time. Figure A.1 depicts the average number of distinct, bypassing tests generated over time.

A.2 Observation

We can observe that in the early stages ML-Driven D finds consistently more bypassing attacks than ML-Driven B does. This turns around only after a certain amount of time is spend in the test execution. From a point in time onwards, ML-Driven B finds consistently more bypassing attacks than ML-Driven D does, and thus the trend of the earlier stages is inverted. Note that this trend is the same for each tested parameter and all repetitions.

This gap in performance is unsatisfactory for several reasons. To maximize the number of bypasses found, the user has to choose between ML-Driven D and ML-Driven B before the test runs

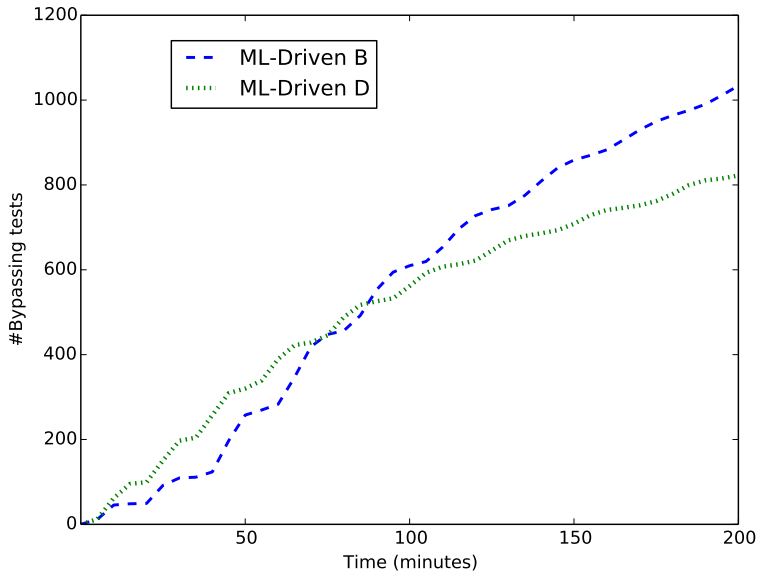


Figure A.1. Average number of bypassing attacks found for ModSecurity (10 repetitions each) with RAN, ML-Driven B, and ML-Driven D. The latter two were run with K = 40%.

starts based on how much time is available. For example, based on the results of Figure A.1, the user should choose ML-Driven D if less than 80 minutes are available, and otherwise ML-Driven B. Moreover, it is unsatisfactory from a scientific standpoint, since both, ML-Driven B and ML-Driven D, implement the same algorithm, only the value assigned to parameter M_{MAX} differs. This raises the question of what causes the difference in effectiveness and if a superior choice of parameters exists, which combines the advantages of ML-Driven D and ML-Driven B.

To answer this question, the difference between ML-Driven D and ML-Driven B has to be reviewed and its impact on the test procedure has to be analysed. As discussed in Section 6.1, the difference between both strategies lies in how the test budget is spent. While ML-Driven D selects only a few candidate attacks for mutation and mutates each candidate more often, ML-Driven B selects more candidate attacks for mutation and mutates each candidate fewer times. As a consequence, both strategies produce the same number of mutants, but ML-Driven D generates more mutants per candidate while ML-Driven B selects more candidate attacks for mutation.

A.3 Analysis

Let $T = [t_{start}, t_{end}]$ be the time interval for the test process and $t_p \in T$ be the point in time after which ML-Driven B finds more bypassing attacks. t_p divides T into two subintervals $[t_{start}, t_p]$, during which ML-Driven D is more effective, and $[t_p, t_{end}]$, during which ML-Driven B is more effective. In the following, both time intervals are examined separately to explain why ML-Driven D or ML-Driven B is more effective during a time interval.

Time interval $[t_{start}, t_p]$. Table A.1 and A.2 show the average bypassing probability of the candidate attacks, which are selected for mutation, for ML-Driven B and ML-Driven D, respectively. For

each iteration, the chosen candidates are separated into a high probability group G_{high}, for which the bypassing probability is higher than 90%, a medium probability group G_{med}, for which the bypassing probability is between 10% and 90%, and a low bypassing probability group G_{low}, for which the bypassing probability is below 10%.

Table A.1. Number of candidate attacks selected for mutation grouped by their bypassing probability for ML-Driven D.

Iteration	G_{low}	G_{med}	G_{high}
1	3.6	0.2	10.5
2	0	0	15.7
3	0	0	9.6
4	0	0	10.3
5	0	0	11
6	0	0	11.8
7	0	0	12.9
8	0	0	13.6
9	0	0	13.3

Table A.2. Number of candidate attacks selected for mutation grouped by their bypassing probability for ML-Driven B.

Iteration	G_{low}	G_{med}	G_{high}
1	78.4	0	10.2
2	50.2	0.5	30.2
3	34.3	0.4	36.4
4	3.7	0.1	60.3
5	0	0	66.2
6	0	0	68.8
7	0	0	73.1
8	0	0	77.3
9	0	0	77.7

For ML-Driven D, in the first iteration 10.5 candidates are in G_{high} and 3.6 candidates are in G_{low}. In the following iterations, all candidates are in G_{high}. For ML-Driven B, in the first iteration 10.2 candidates are in G_{high} and 78.4 candidates are in G_{low}. In the following iterations, the number of candidates in G_{high} increases in the second iteration to 30.2, in the third iteration to 36.4, and in the fourth iteration to 60.3 while the number of candidates in G_{low} decreases in the second iteration to 50.2, in the third iteration to 34.3, and in the fourth iteration to 3.7. Starting from the fifth iteration, all candidates are in G_{high}. In summary, ML-Driven D selects attacks for mutation almost exclusively from G_{high} while ML-Driven B selects in the first three iterations a significant number of attacks from G_{low}.

To assess how the selected candidate attacks influence the number of bypassing attacks, we examine which candidates are successful in generating bypassing mutants. Table A.3 and A.4 show the number of blocked and bypassing mutants separated by group to which the candidate attack belongs to. For ML-Driven D, 58.3 bypassing mutants are generated from candidates belonging to G_{high} and 1.5 bypassing mutants are generated from candidates belonging to G_{low}. Similarly, 2276.8 blocked

Table A.3. Number of bypassing and blocked mutants generated from candidates belonging to G_{low} and G_{high} for ML-Driven D.

Iteration	Bypassing		Blocked	
	G_{low}	G_{high}	G_{low}	G_{high}
1	1.5	58.3	944.2	2276.8
2	0	61.9	0	3799.8
3	0	93	0	3667.8
4	0	82.4	0	3604.2
5	0	82.9	0	3589.8
6	0	56.1	0	3649.2
7	0	48.8	0	3601.8
8	0	31.4	0	3639.8
9	0	38.5	0	3744.9

Table A.4. Number of bypassing and blocked mutants generated from candidates belonging to G_{low} and G_{high} for ML-Driven B.

Iteration	Bypassing		Blocked	
	G_{low}	G_{high}	G_{low}	G_{high}
1	5.7	24.5	3097.3	338.6
2	2.6	29.6	2706.9	759.9
3	2.6	62.6	2061	1452.4
4	0.1	134.7	230.5	3028.5
5	0	125.4	0	3281.4
6	0	95.3	0	3294.1
7	0	93.6	0	3294
8	0	79.6	0	3347.1
9	0	55.5	0	3370.4

mutants are from candidates belonging to G_{high} and 944.2 mutants are from candidates belonging to G_{low}. In the following iterations, all candidates are selected from G_{high}. For ML-Driven B, 24.5 bypassing mutants are generated from candidates belonging to G_{high} and 5.7 are generated from candidates belonging to G_{low}. Similarly, 338.6 blocked mutants are generated from candidates belonging to G_{high} and 3097.3 mutants are from candidates belonging to G_{low}.

To conclude, the large majority of bypassing attacks is generated as expected from candidates belonging to G_{high} and attacks generated from G_{low} are likely to be blocked. More interestingly, the presented data underlines that ML-Driven D is more successful at the beginning than ML-Driven B because it spends the test budget to a significantly larger fraction on candidate attacks from G_{high} and thus generates more bypassing mutants.

Based on the presented analysis we find that during the early stages of the test process only a small number of candidate attacks have a high probability of bypassing the firewall. ML-Driven D is more effective during this stage of the test process, because the strategy spends the mutation budget on the few known candidates that have a high likelihood of bypassing the firewall and is then more likely to generate bypassing attacks. On the other hand, ML-Driven B is less effective in the earlier stages of the test process because the strategy spends more mutation budget on candidate attacks that have a

lower likelihood of bypassing the firewall.

Time interval $[t_p, t_{end}]$. From the fifth iteration onwards both strategies select candidate attacks from G_{high} only. Nevertheless, ML-Driven B surpasses ML-Driven D in the number of bypassing attacks, hence there must be an additional factor influencing effectiveness. To analyse this, we introduce the measure of *mutation efficiency*. Let M_B be the number of bypassing mutants and M_T the total number of mutants that are generated from a candidate attack, then mutation efficiency M_{eff} is the ratio M_B/M_T.

Table A.5 and A.6 show the mutation efficiency per iteration for ML-Driven B and ML-Driven D. Starting from iteration 4, when both strategies select candidates almost exclusively from G_{high}, the mutation efficiency is 0.04 for ML-Driven B and only 0.02 for ML-Driven D. Similarly, the mutation efficiency in all following iterations is higher for ML-Driven B than for ML-Driven D.

Table A.5. Mutation efficiency for ML-Driven B.

Iteration	M_B	M_T	M_{Eff}
1	0.34	38.78	0.01
2	0.40	43.00	0.01
3	0.92	49.74	0.02
4	2.10	50.91	0.04
5	1.89	49.57	0.04
6	1.39	47.88	0.03
7	1.28	45.06	0.03
8	1.03	43.30	0.02
9	0.71	43.38	0.02

Table A.6. Mutation efficiency for ML-Driven D.

Iteration	M_B	M_T	M_{Eff}
1	4.21	227.45	0.02
2	3.95	242.15	0.02
3	9.67	381.45	0.03
4	7.99	349.66	0.02
5	7.54	326.34	0.02
6	4.77	310.07	0.02
7	3.78	278.71	0.01
8	2.31	267.33	0.01
9	2.90	281.41	0.01

To conclude, ML-Driven B in comparison to ML-Driven D generates fewer bypassing mutants per candidate, but it also generates less mutants per candidate in total. For ML-Driven B, the ratio of bypassing mutants divided by the total number of mutants is higher, hence mutation efficiency is favourable to ML-Driven B which surpasses ML-Driven D in later iterations.

The analysed data suggests that distributing the mutation budget equally between candidates from G_{high} is more efficient than focusing the budget on a few G_{high} candidates.

Appendix B

Testing Web Application Firewalls: Test Results Per Parameter

This appendix lists the evaluation results of the WAF testing technique in more detail (see Chapter 6). Due to space constrains in Chapter 6, we divided test runs with a similar number of bypassing attacks into groups and computed the average. This appendix presents the results of each test run separately.

The figures are organised as follows:

- The **figure on the left** shows the average number of bypassing tests found for a tested operation.
- The **figure on the right** shows the statistical variation for the data in the left figure (10 repetitions).

B.1 Results for ModSecurity

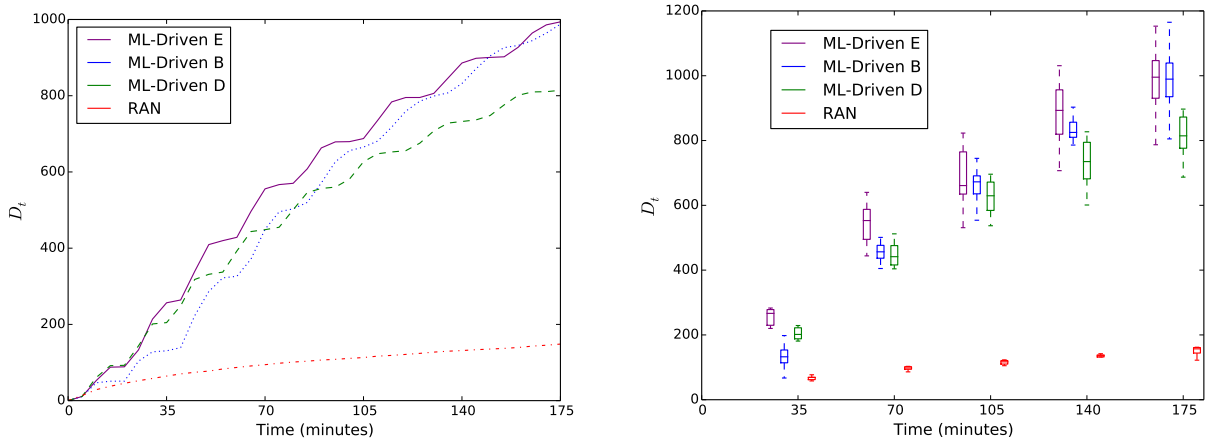


Figure B.1. Test result for operation *confirmRoom*.

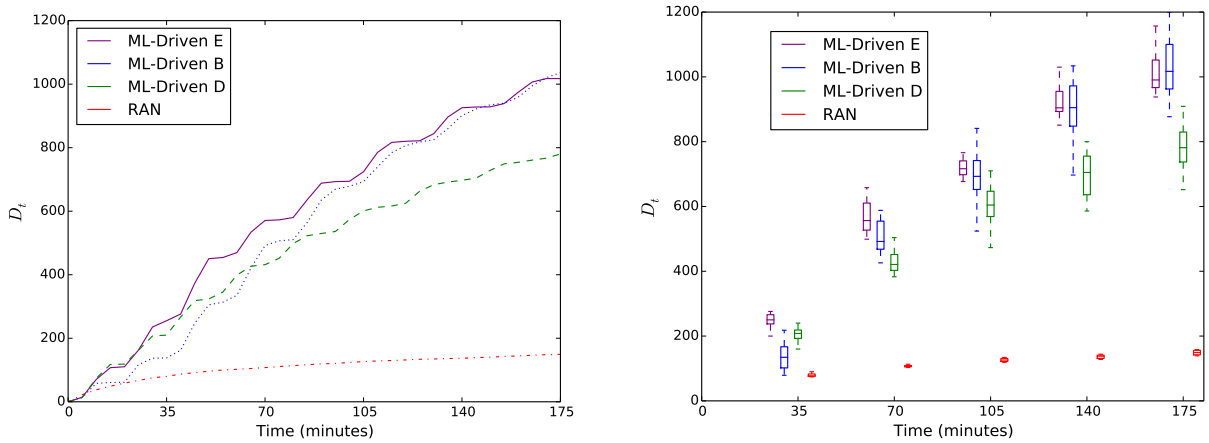


Figure B.2. Test result for operation *getCustomerByID*.

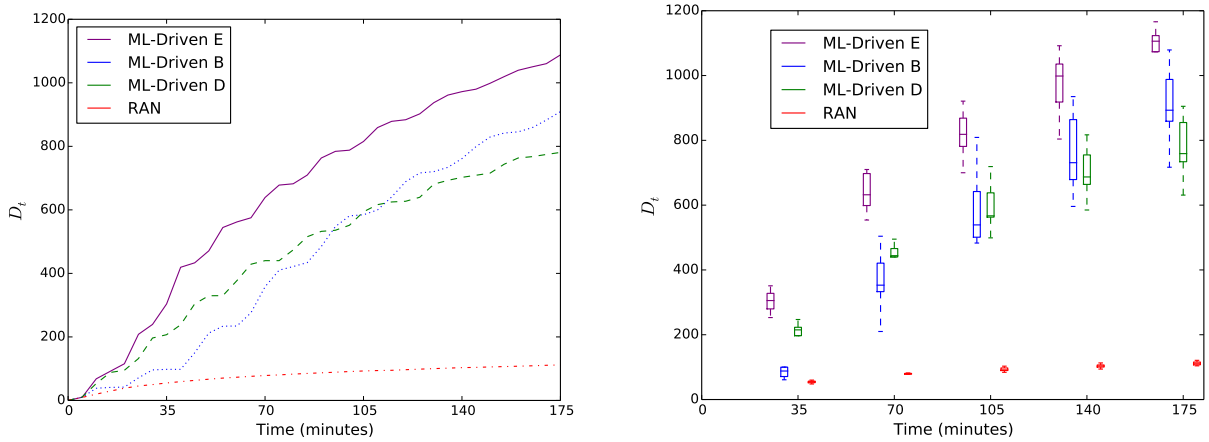


Figure B.3. Test result for operation *doPayment*.

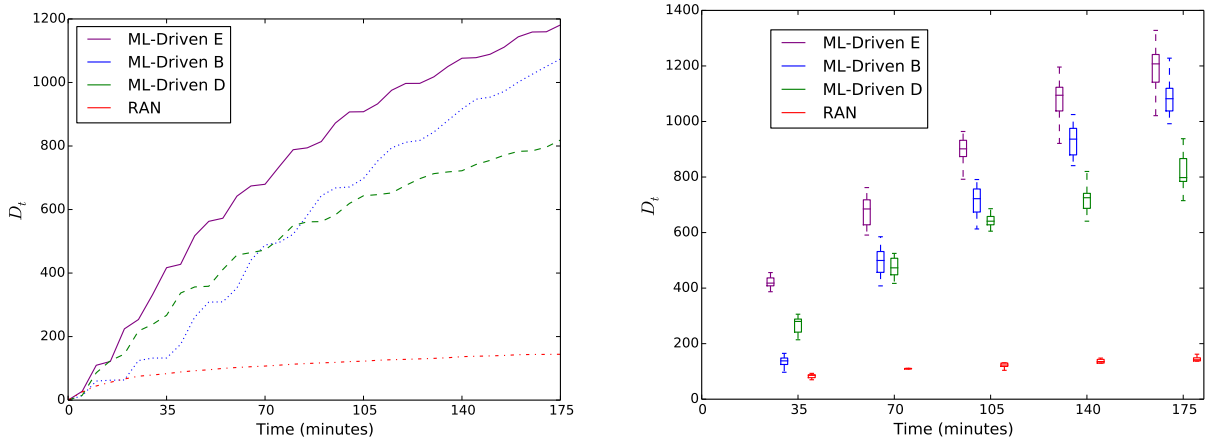


Figure B.4. Test result for operation *expireTicket*.

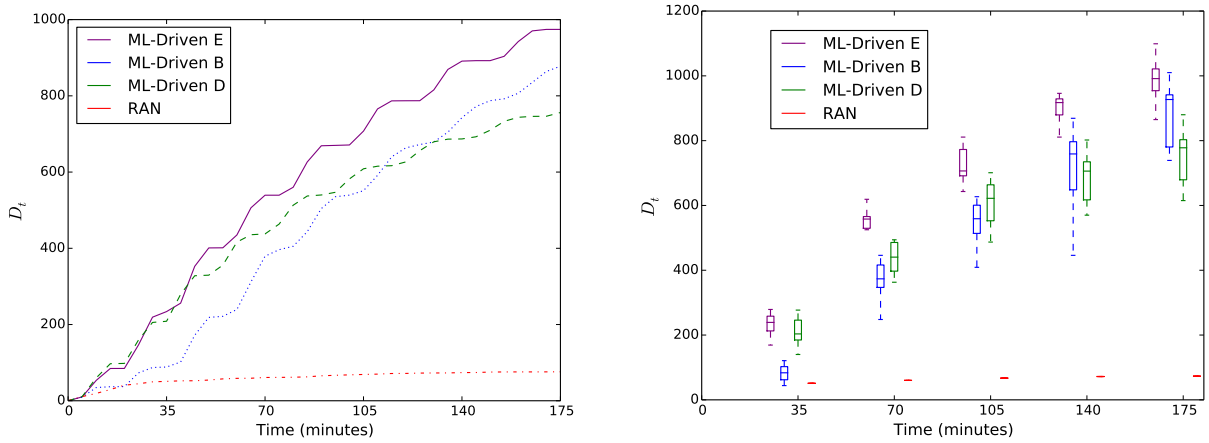


Figure B.5. Test result for operation *searchByModule*.

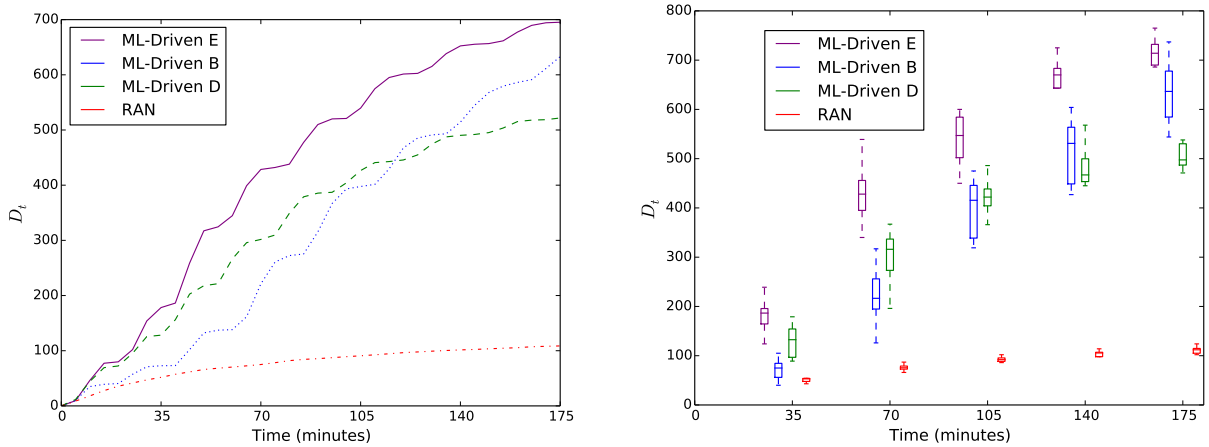


Figure B.6. Test result for operation *getEntries*.

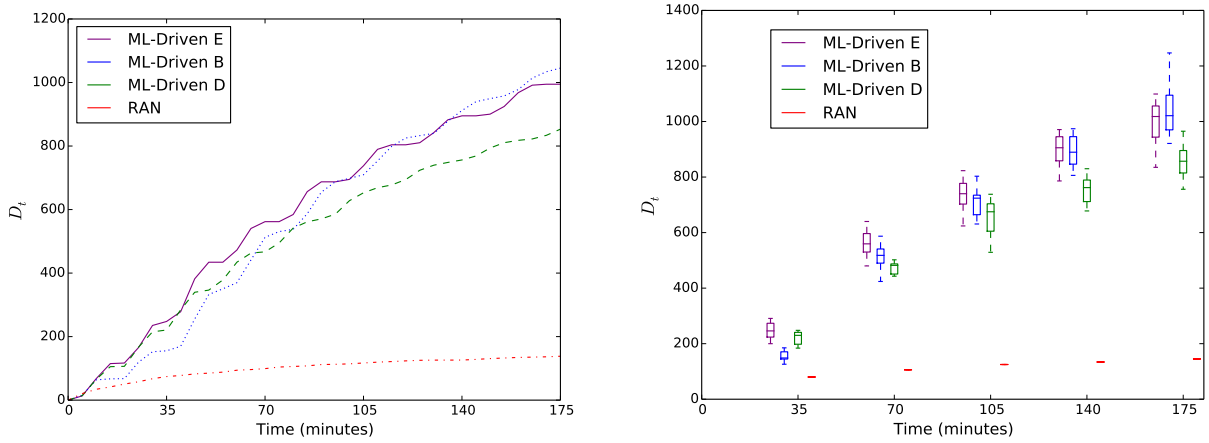


Figure B.7. Test result for operation *getRelationships*.

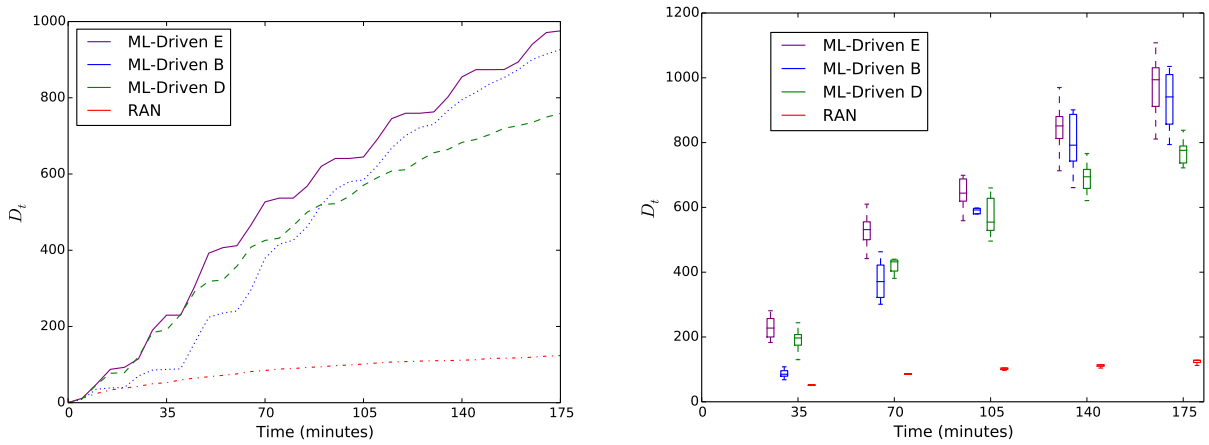


Figure B.8. Test result for operation *simulatePayment*.

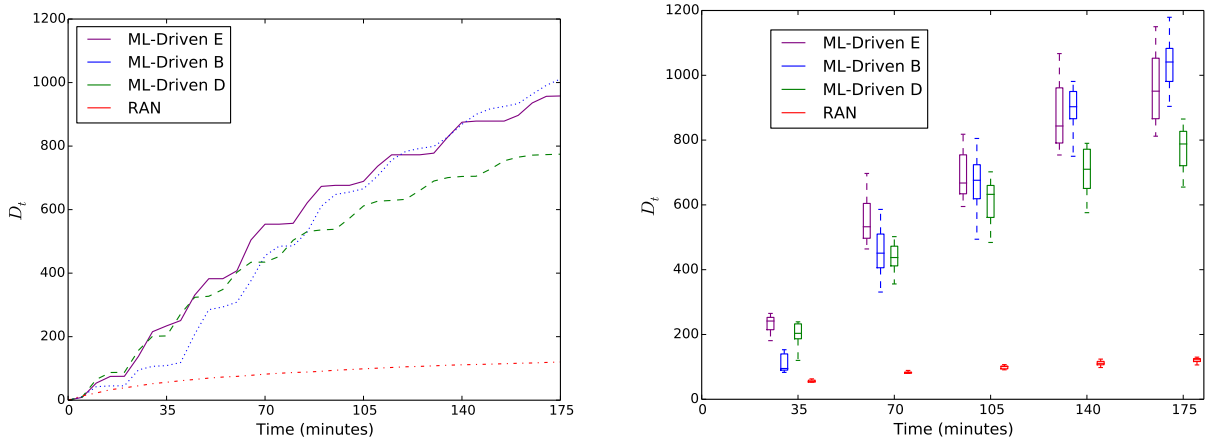


Figure B.9. Test result for operation *setEntry*.

B.2 Results for a proprietary WAF

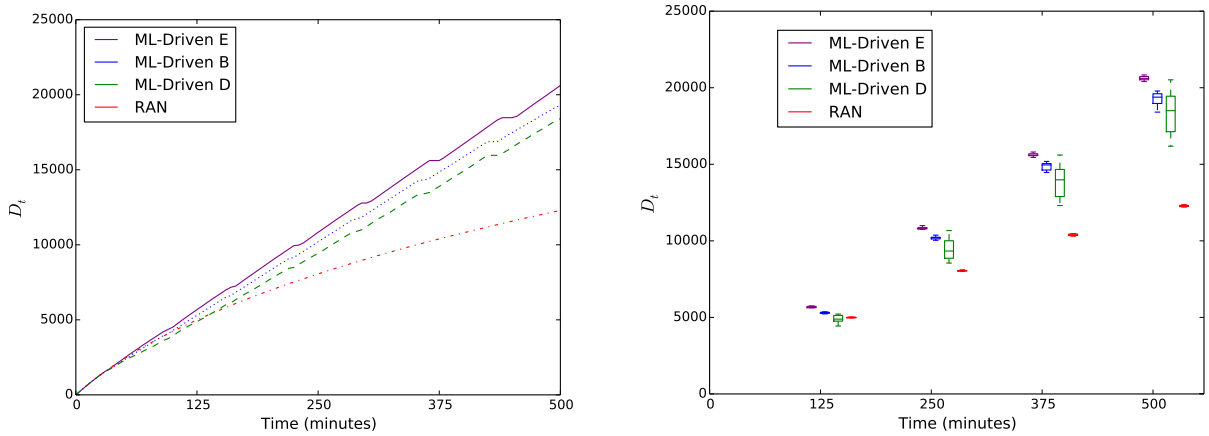


Figure B.10. Test result for operation *AddressLine2*.

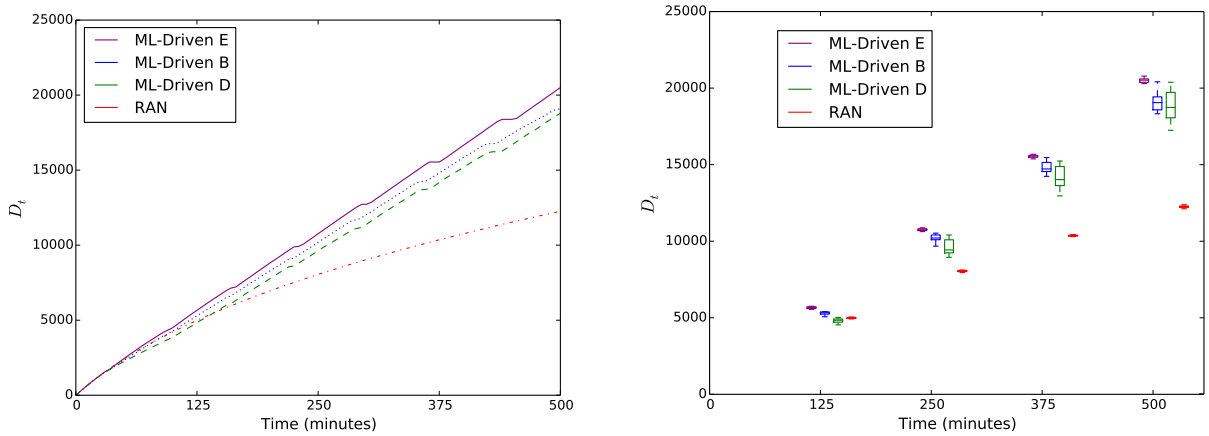


Figure B.11. Test result for operation *AddressLine3*.

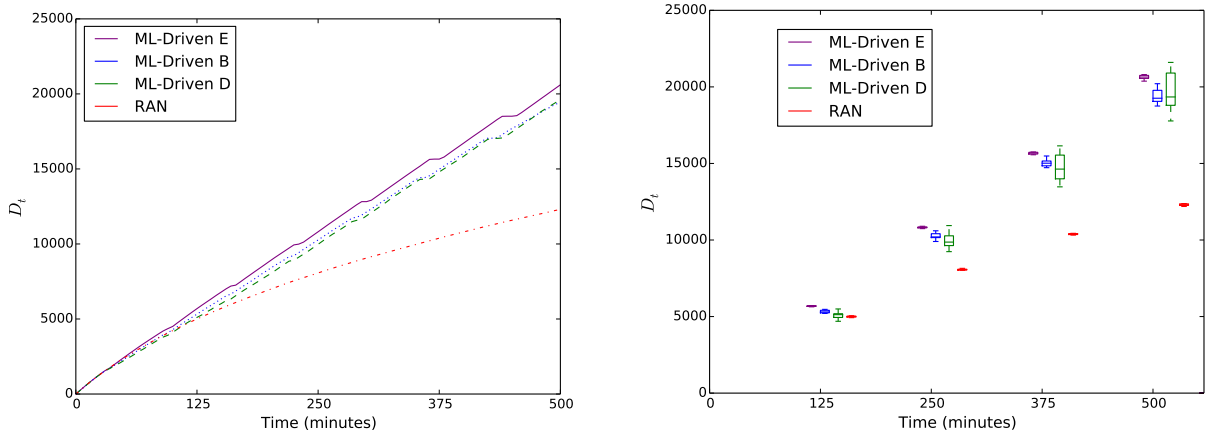


Figure B.12. Test result for operation *AddressLine4*.

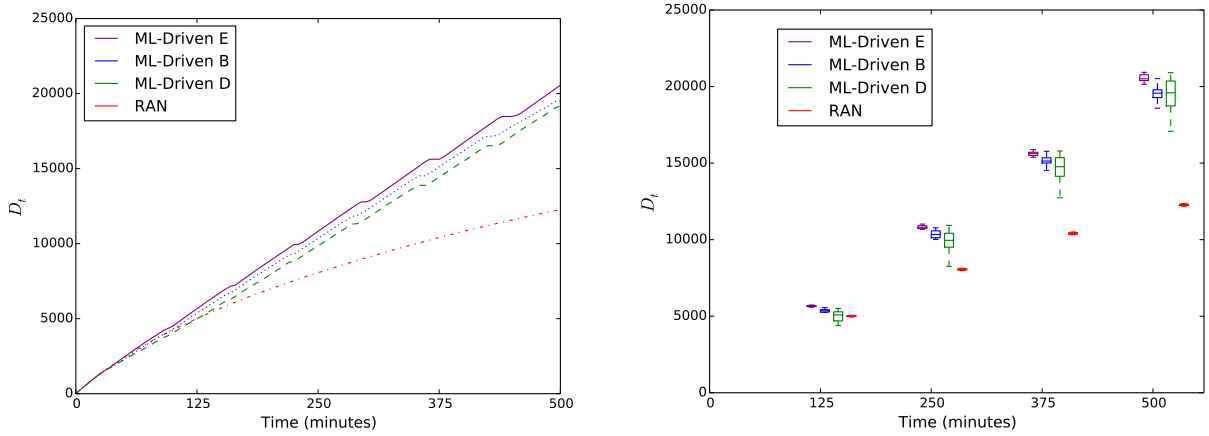


Figure B.13. Test result for operation *AddressLine5*.

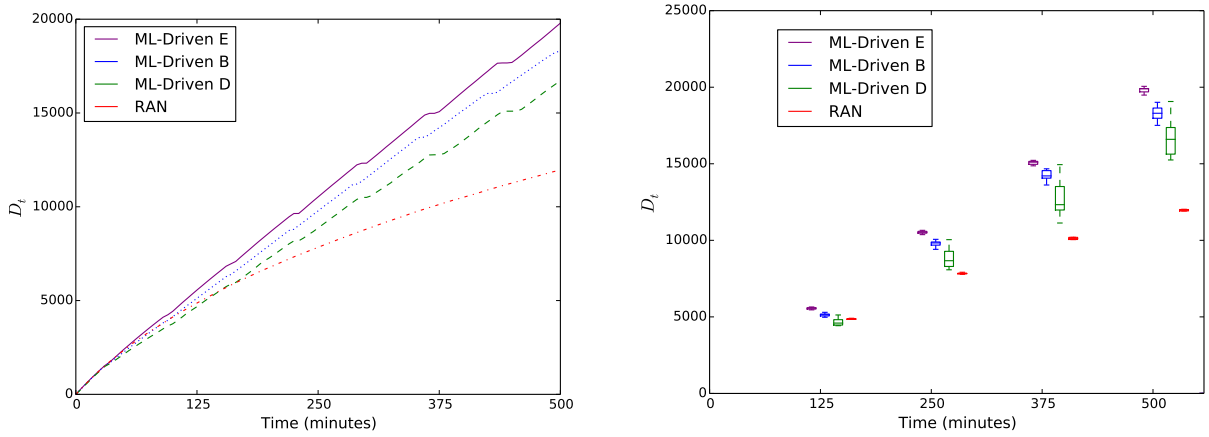


Figure B.14. Test result for operation *BankAccount*.

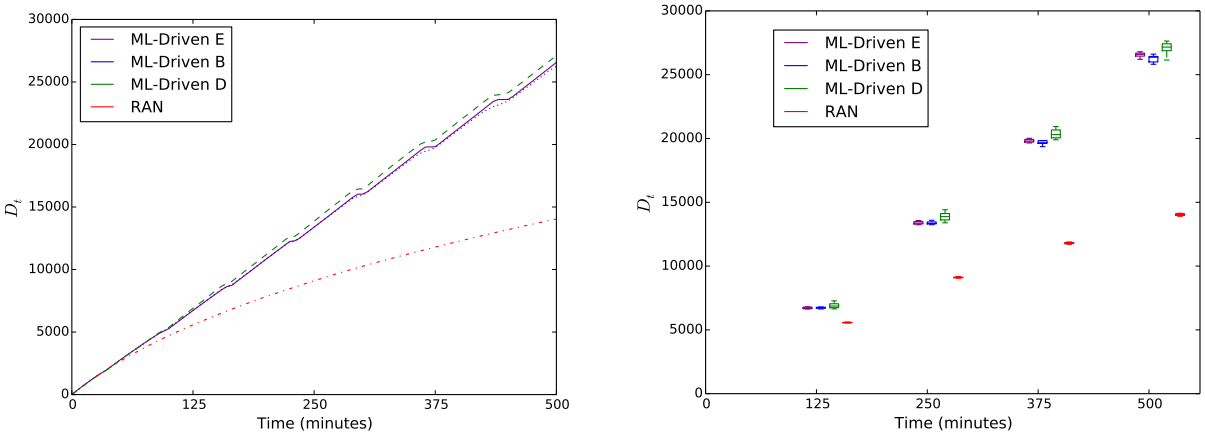


Figure B.15. Test result for operation *BankReference*.

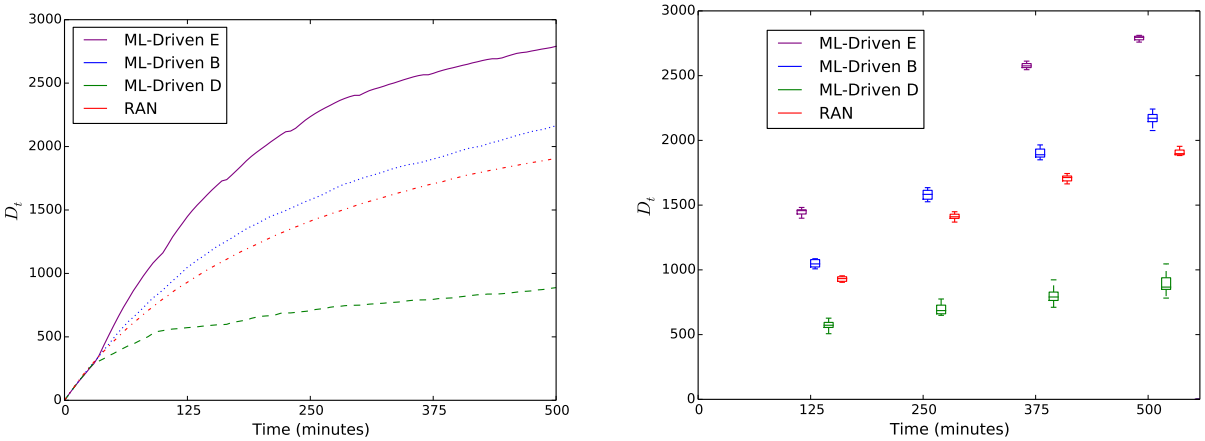


Figure B.16. Test result for operation *ClearingRef*.

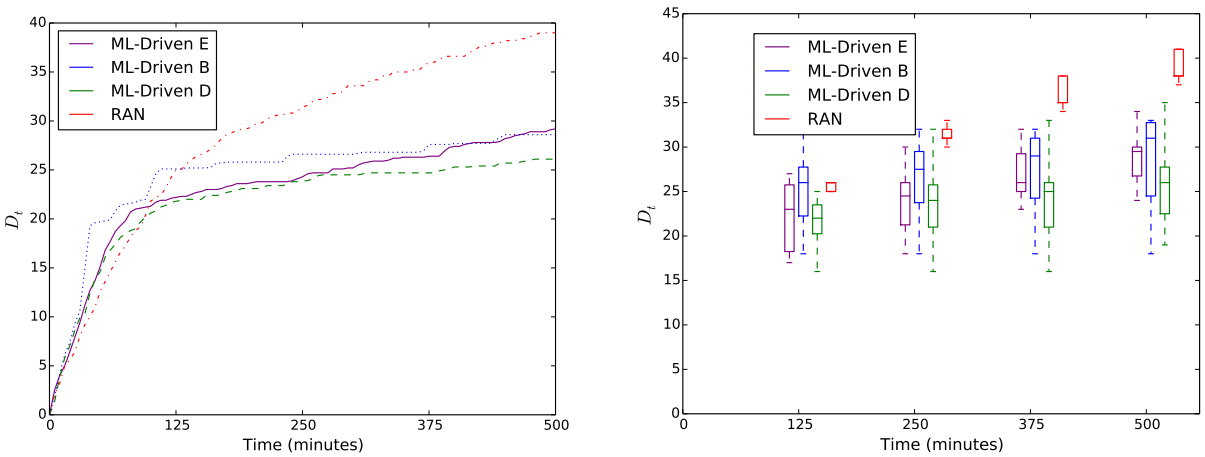


Figure B.17. Test result for operation *ClientClass*.

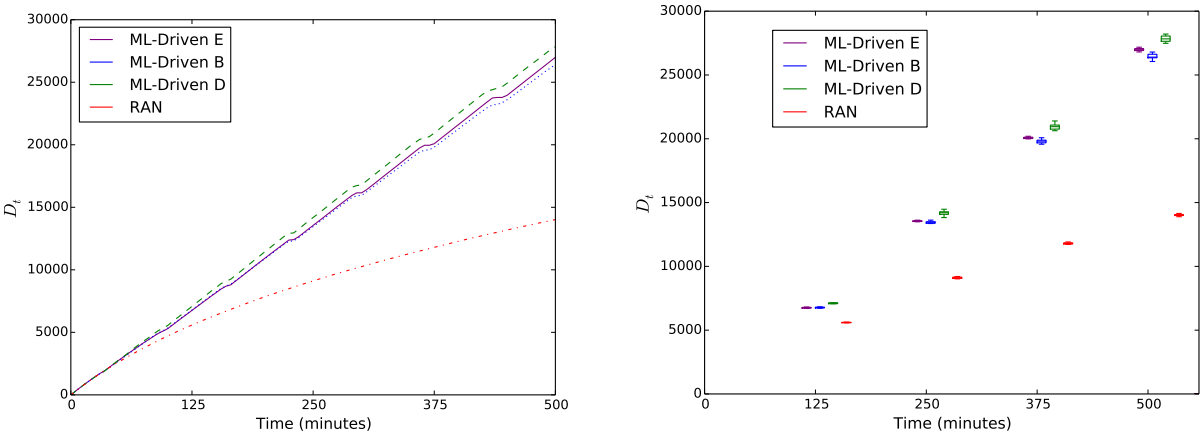


Figure B.18. Test result for operation *Email*.

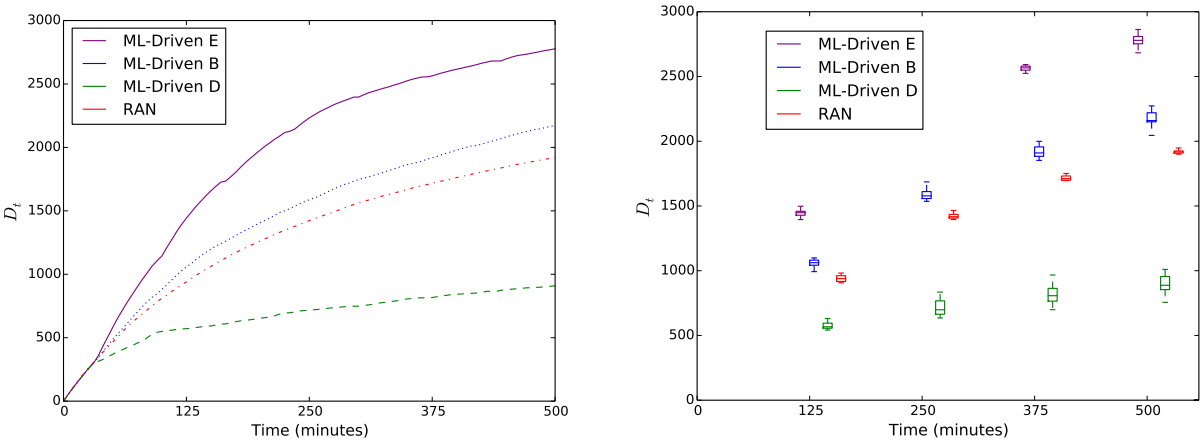


Figure B.19. Test result for operation *FaxNumber*.

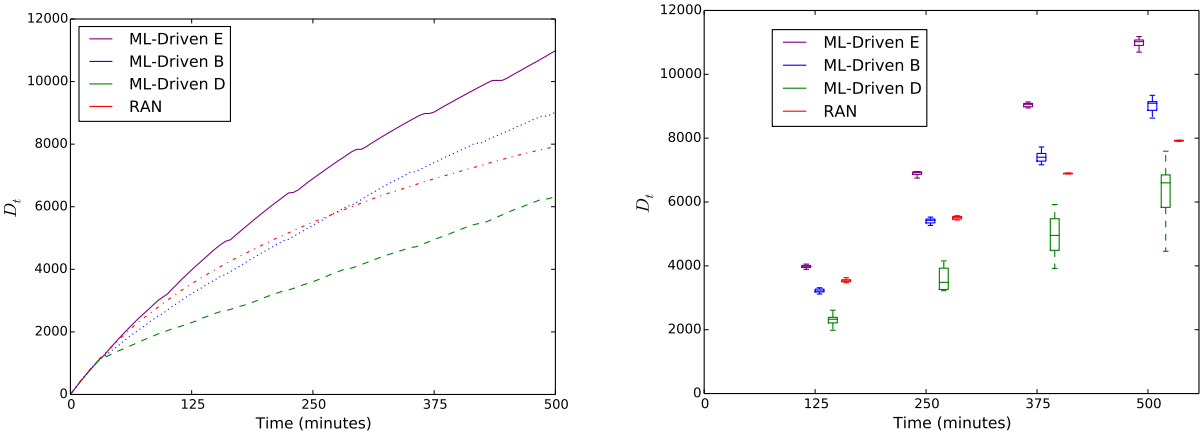


Figure B.20. Test result for operation *FirstName*.

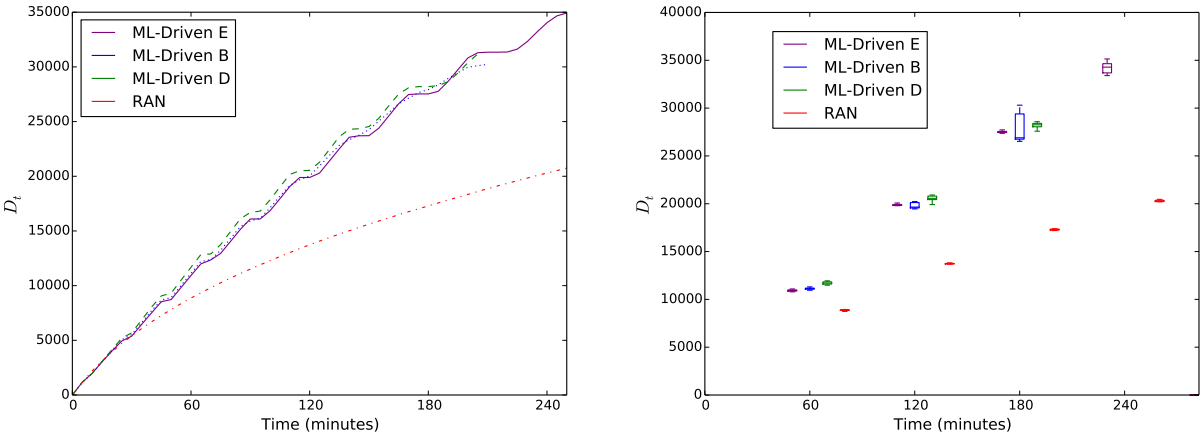


Figure B.21. Test result for operation *Label*.

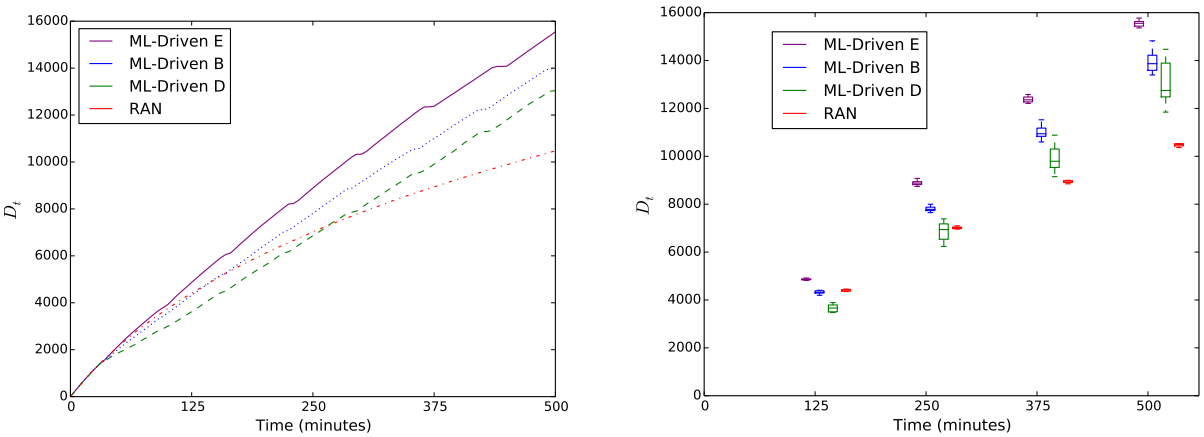


Figure B.22. Test result for operation *LastName*.

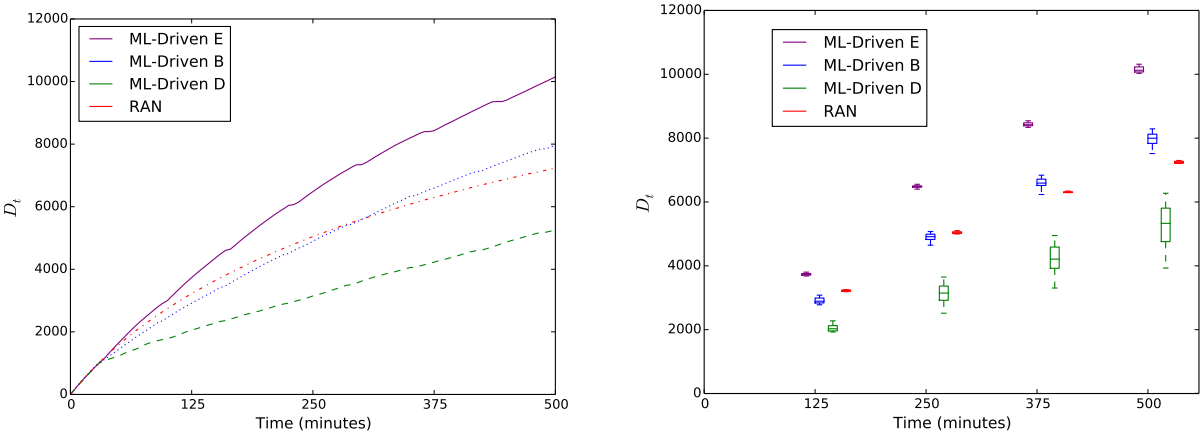


Figure B.23. Test result for operation *Line4*.

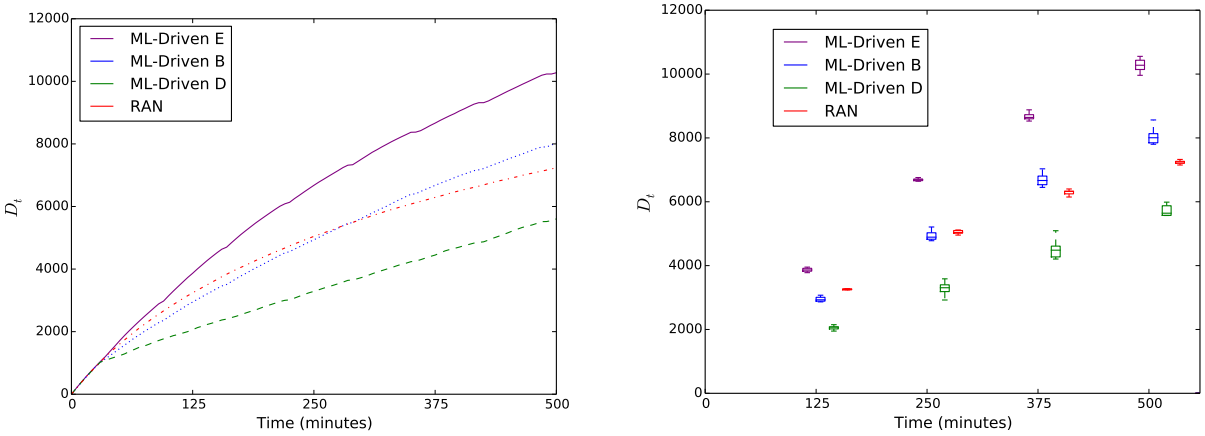


Figure B.24. Test result for operation *Line5*.

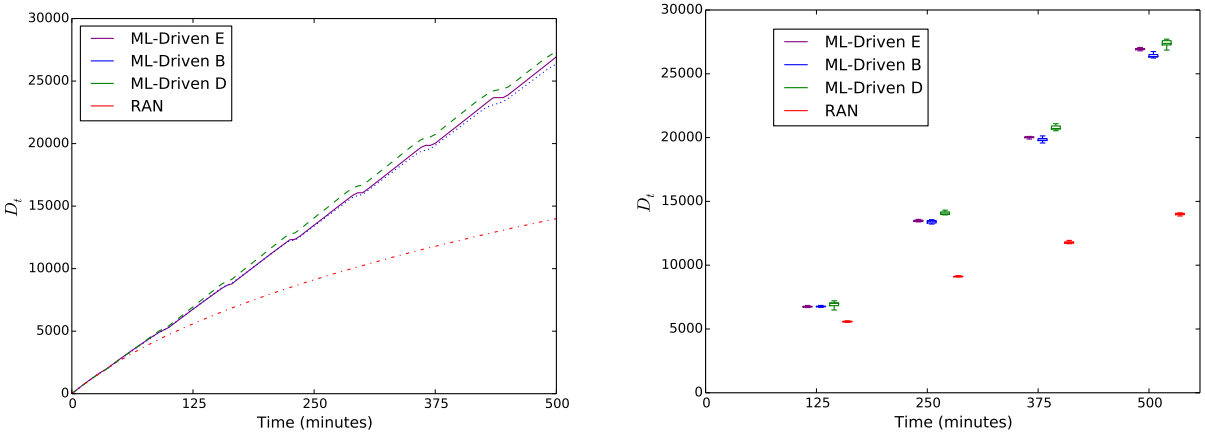


Figure B.25. Test result for operation *Locality*.

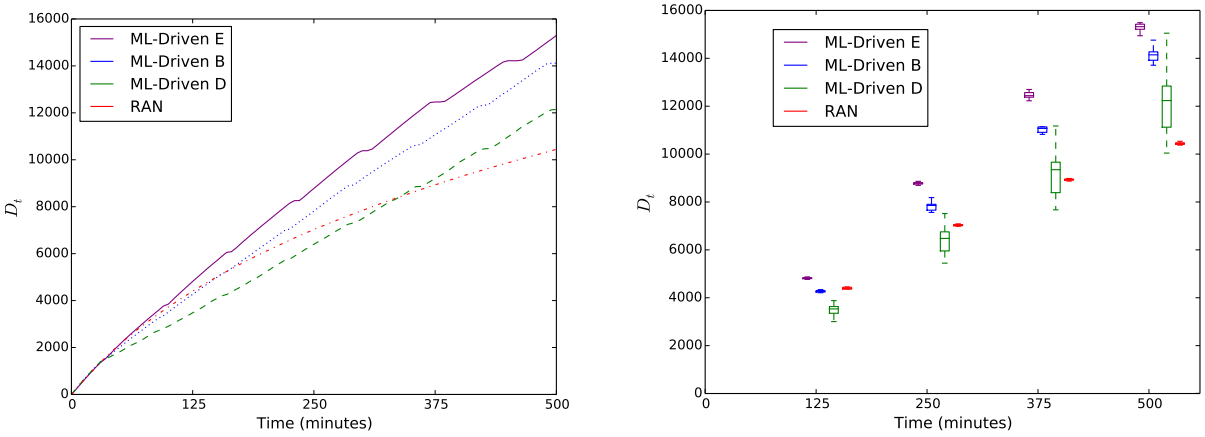


Figure B.26. Test result for operation *MaidenName*.

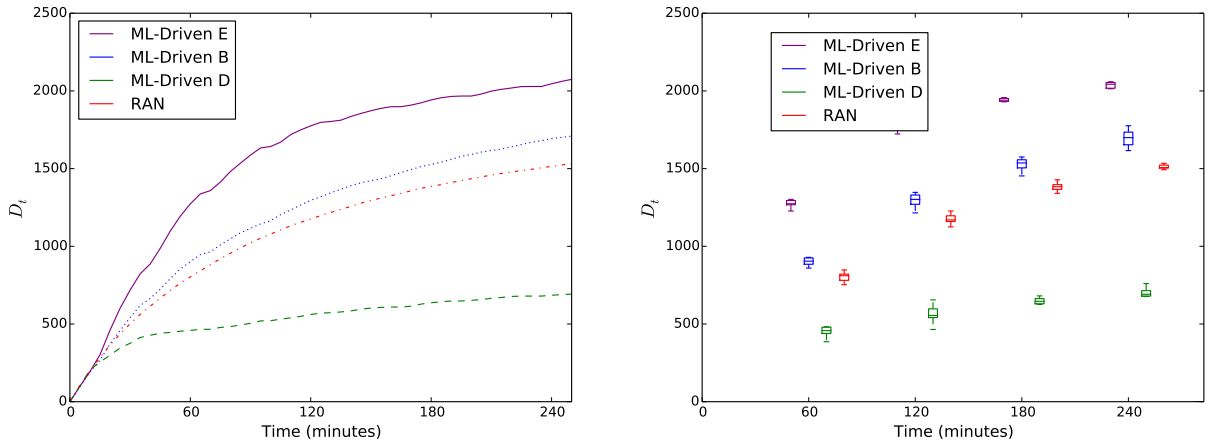


Figure B.27. Test result for operation *MerchantId*.

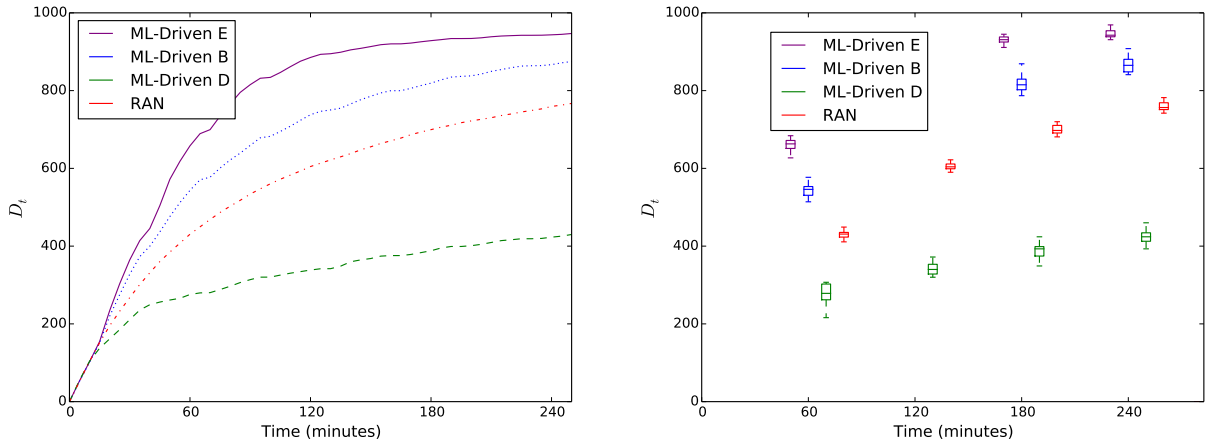


Figure B.28. Test result for operation *MerchantLocality*.

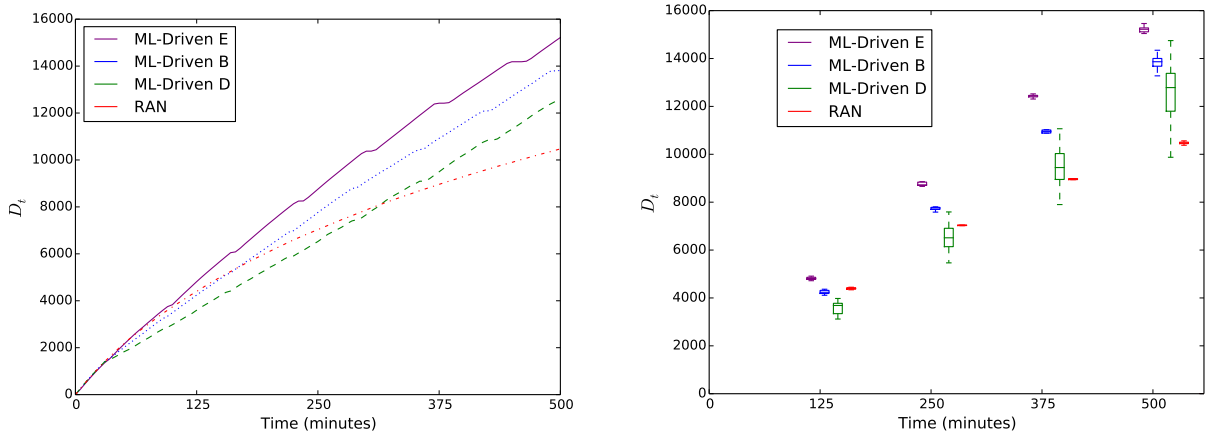


Figure B.29. Test result for operation *MotherName*.

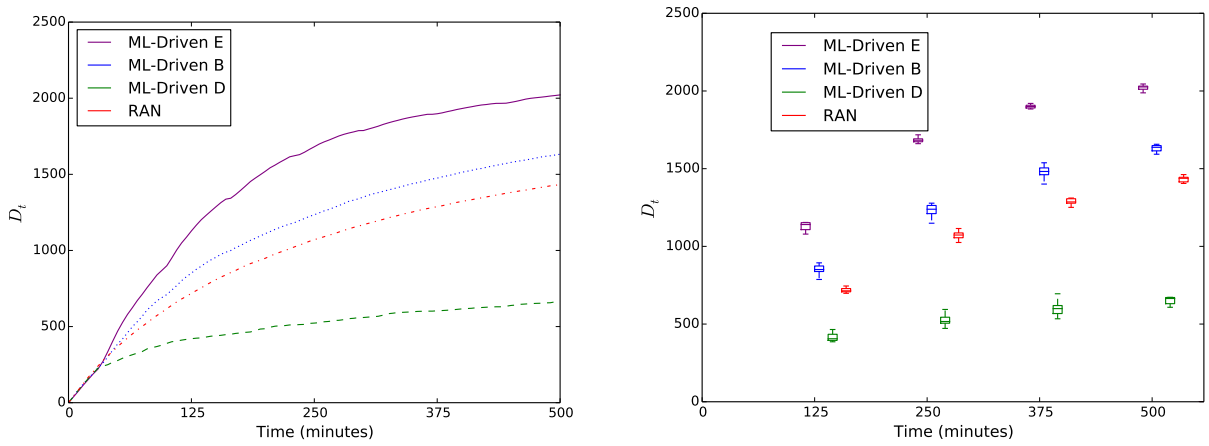


Figure B.30. Test result for operation *Passeport*.

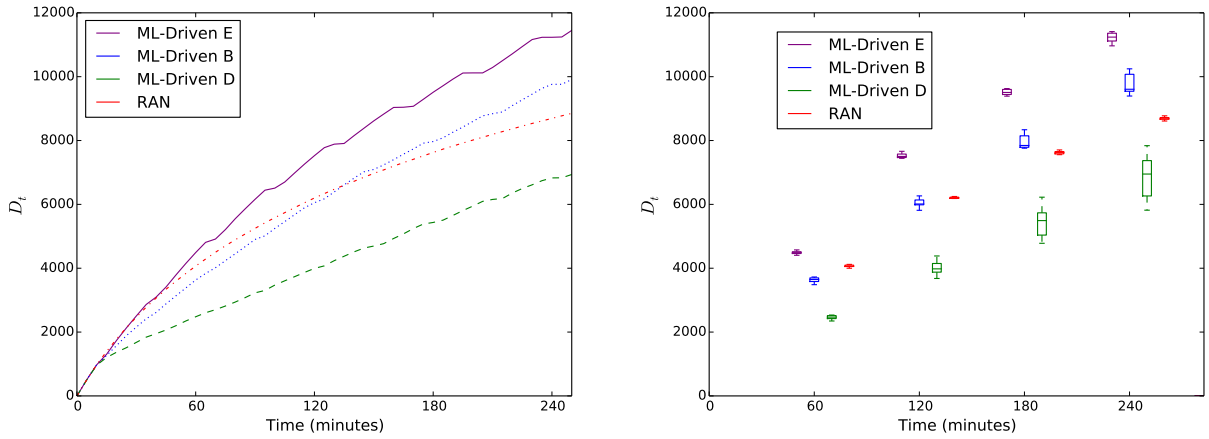


Figure B.31. Test result for operation *PaymentOption*.

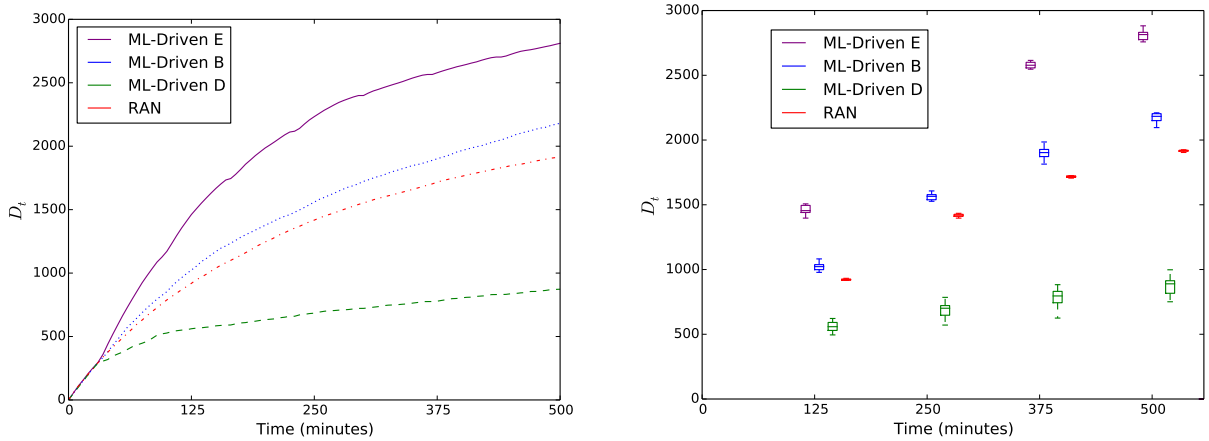


Figure B.32. Test result for operation *PhoneNumber*.

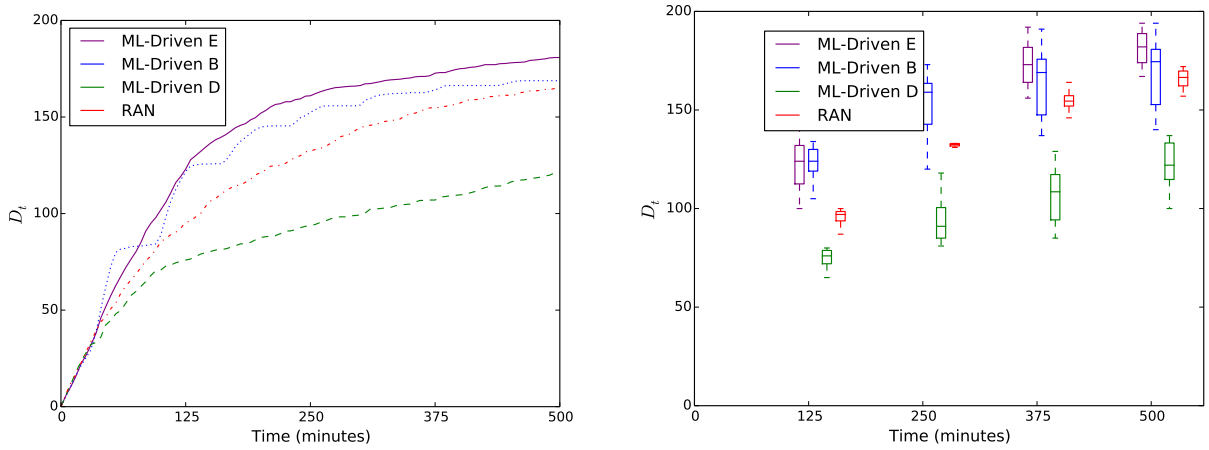


Figure B.33. Test result for operation *PostalCode*.

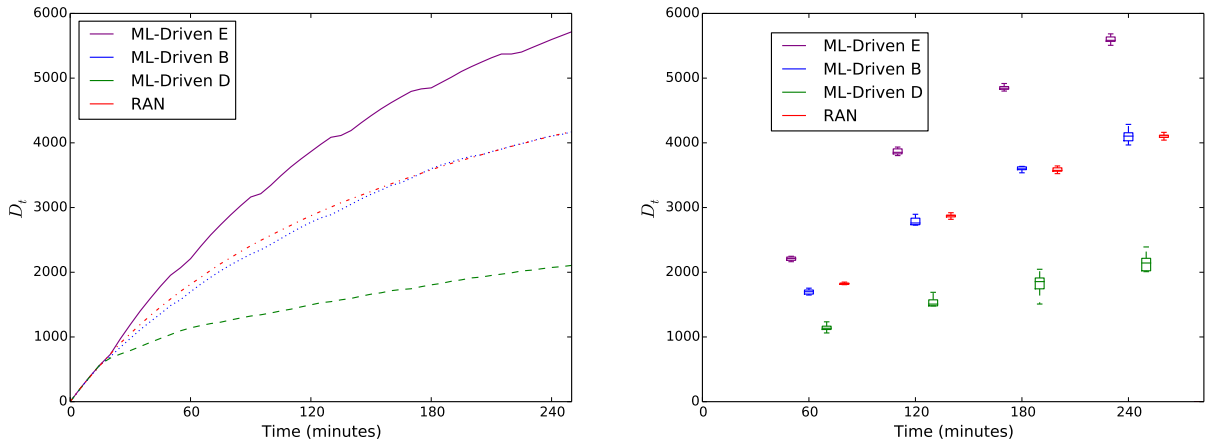


Figure B.34. Test result for operation *RequestId*.

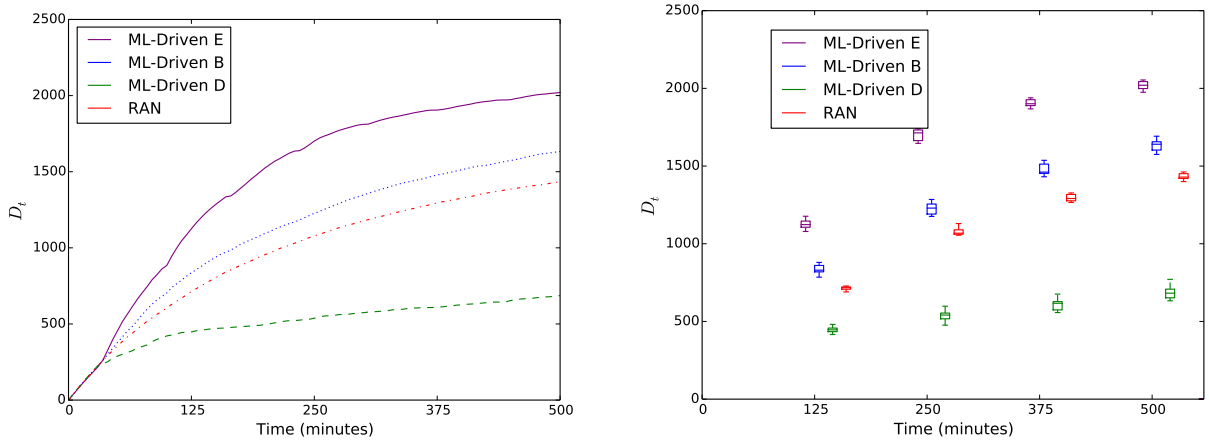


Figure B.35. Test result for operation *SocialNumber*.

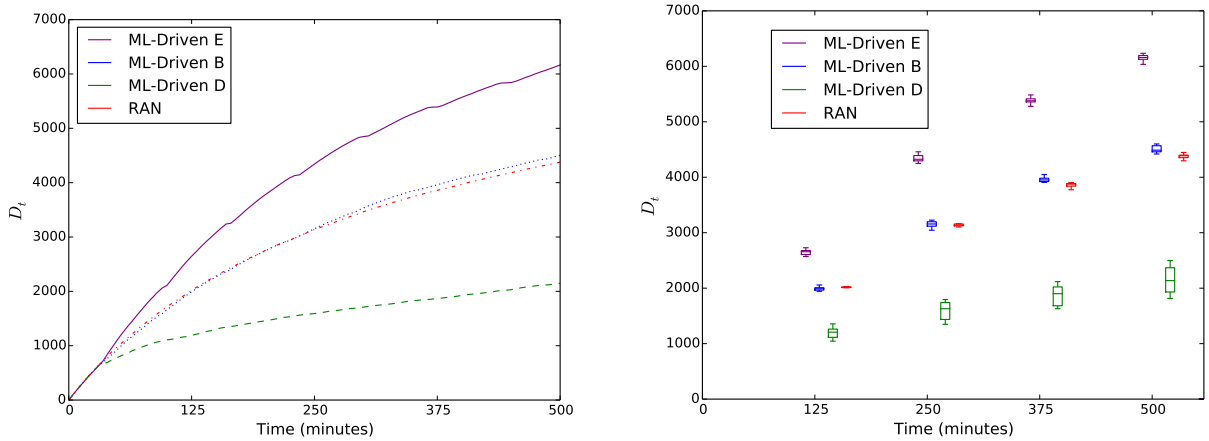


Figure B.36. Test result for operation *Title*.

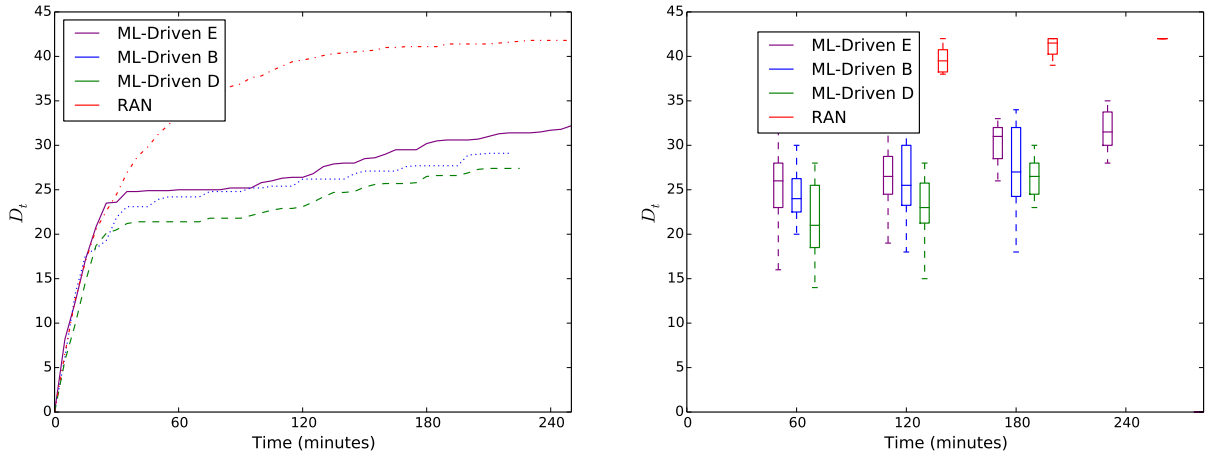


Figure B.37. Test result for operation *UserName*.